THE UNIVERSITY *of* EDINBURGH

Parallel solution of linear programs

Edmund Smith

October 26, 2013

First supervisor: *Dr Julian Hall*
Second supervisor: *Dr Andreas Grothey*

Doctor of Philosophy
University of Edinburgh
2013

Contents

I dedicate this thesis to my grandmother,

Edith Holding (1921-2012).

Chapter 1

Introduction

1.1 Overview

The factors limiting the performance of computer software periodically undergo sudden shifts, resulting from technological progress, and these shifts can have profound implications for the design of high performance codes. At the present time, the speed with which hardware can execute a single stream of instructions has reached a plateau. It is now the number of instruction streams that may be executed concurrently which underpins estimates of compute power, and with this change, a critical limitation on the performance of software has come to be the degree to which it can be parallelised.

The research in this thesis is concerned with the means by which codes for linear programming may be adapted to this new hardware. For the most part, it is codes implementing the simplex method which will be discussed, though these have typically lower performance for single solves than those implementing interior point methods. However, the ability of the simplex method to rapidly re-solve a problem makes it at present indispensable as a subroutine for mixed integer programming.

The long history of the simplex method as a practical technique, with applications in many industries and government, has led to such codes reaching a great level of sophistication. It would be unexpected in a research project such as this one to match the performance of top commercial codes with many years of development behind them. The simplex codes described in this thesis are, however, able to solve real problems of small to moderate size, rather than being confined to random or otherwise artificially generated instances.

The remainder of this thesis is structured as follows. The rest of this chapter gives a brief overview of the essential elements of modern parallel hardware and of the linear programming problem. Both the simplex method and interior point methods are discussed, along with some of the key algorithmic enhancements required for such systems to solve real-world problems. Some background on the parallelisation of both types of code is given.

The next chapter describes two standard simplex codes designed to exploit the current generation of hardware. `i6` is a parallel standard simplex solver capable of being applied to a range of real problems, and showing exceptional performance for dense, square programs. `i8` is also a parallel, standard simplex solver, but now implemented for graphics processing units (GPUs).

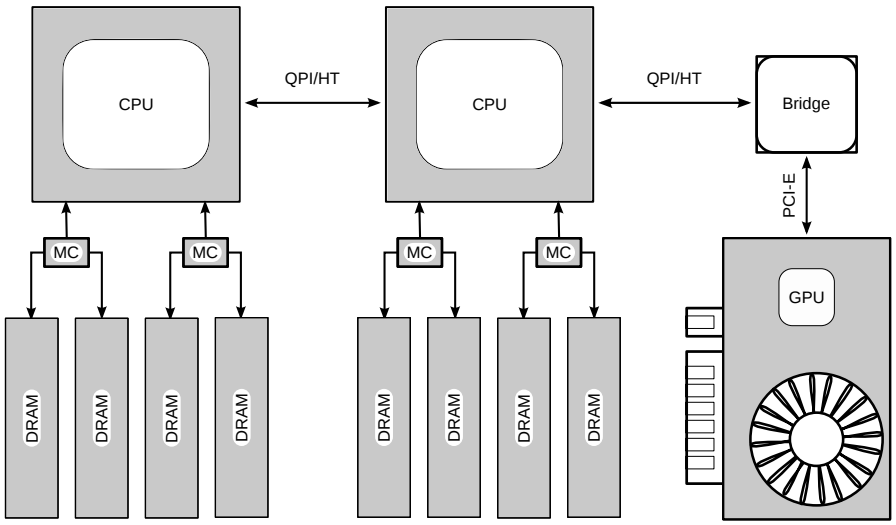


Figure 1.1: A schematic of a typical commodity workstation (c. 2012). Note that the memory controllers (MC) are integrated into the CPUs, but are shown separately here for clarity.

`i7` is the major focus of this thesis, and is covered in the next chapter. It is a parallel, revised simplex solver for primal block-angular problems, which can automatically convert many general programs to this form. Such problems have a particular structure to their basis matrices, and it is this observation which is used to enable parallelism, rather than a more common column- or cut-generation mechanism.

The final chapter describes `hmf`, which is based on the HOPDM interior point library [7, 49]. It provides accelerated parallel linear algebra for general problems, and also routines for quadratic assignment problems which do not require the constraint matrix to be formed explicitly.

In the appendices, a selection of simplex subsystems are subjected to deeper analysis. This includes some sections, including an algorithm for dual phase one, and a primal ratio test, which are believed to be novel.

1.2 Parallel commodity hardware

The principal driver for this work is a change in the characteristics of the computer hardware on which a solver for linear programming will run. Some of those characteristics are now described, in order to give context to the discussion in later chapters.

The focus in this thesis is on codes for a single workstation, having a small number of processor packages, each containing multiple cores. This is a very common and affordable type of workstation at the present time. Both compute clusters and distributed networks of machines are intentionally excluded from consideration by this definition, the former due to their high cost, and the latter owing to the impossibility of rapid communication between the constituent parts.

1.2.1 Main memory

Main memory is the space available on a system for programs and working data. The installed capacity of main memory has increased much more rapidly than the size of typical linear programming problems, to the extent that holding multiple copies of the required working data in memory, arranged in different ways, is now practicable. Indeed, for all but the largest problems, the memory available to a solver is essentially infinite. This in turn means that principal storage is no longer, for linear programming, a key hardware component.

In most current systems, main memory takes the form of *random access memory* (RAM), or more particularly SDRAM (synchronous dynamic RAM). This is installed in *modules*, vast arrays of transistor-based cells which can hold charge only for a short time - they "leak". The temporary nature of DRAM storage requires that each of its elements be refreshed at regular intervals, but while this is being carried out, a cell will be unavailable for use.

In order to read from, or write to, a particular memory location, a relatively complex, multi-stage process must be invoked. Within each module, memory cells are arranged into rows and columns, and the particular row and column of interest must be separately selected before a read request can be initiated. Management of DRAM is now typically overseen by dedicated *memory controllers* built into each processor package.*

Memory is clocked at a relatively low frequency compared to the processors, which means that memory accesses happen natively at a far lower rate than processor operations. This affects both *memory latency*, the fixed cost time to complete any memory access, and *memory bandwidth*, the maximum transfer rate.

SDRAM memory modules allow requests to be pipelined, with the next request beginning before the previous one has completed. DDR (double data-rate) memory enables multiple bits to be accessed per memory clock cycle, and per bus line, by a clever scheme of interleaving, but this in turn creates a *minimum burst* of memory which must be read. A dual channel memory controller will map alternating runs of memory addresses to each of a pair of modules, allowing memory accesses to be serviced from both simultaneously.

The outcome of these, and other, technologies is that memory accesses near to one another incur less overhead from addressing, and that sequential access may benefit from additional burst acceleration. There is little protection at this level, however, from latency, so that any read direct from main memory can be expected to be extremely expensive. There is significantly more detail on these topics in [34].

1.2.2 Cache

Caches are small memories, separate from main memory, which hold code and data for immediate use. They are usually located physically on the same silicon die as the processors, and constructed from *static random access memory* (SRAM), which is much faster and more expensive than SDRAM, and has much higher power consumption.

*Until recently, the memory controller was part of a separate unit, called the Northbridge, but this design does not scale up as the number of processors increases.

The processor caches provide a transparent mirror of main memory, invisible to the program, such that all memory accesses are serviced from them, and only when a memory location is not already cached is a request issued to main memory. Each cache is made up of *lines*, fixed length blocks of storage that must be read from or written to RAM at the same time, and each of which holds the data from a set of contiguous addresses.

The particular addresses a cache line holds will vary during the execution of a program, so that to perform a memory access, a processor must determine the relevant line of cache which either holds, or will hold, the requested location. In a *direct-mapped cache*, there is only one line which could hold each address, so that determining whether it is already in cache can be done quickly, but two parts of memory may end up competing for the same line. In a *fully associative cache*, every line may represent any part of memory, but finding the right line for each operation becomes correspondingly more expensive. Rather than these extremes, most caches are *set associative*, where each part of memory may map to some small number of cache lines (say between 2 and 16, depending on the particular processor and cache).

Caches are organised in a hierarchy, from L1, with the highest bandwidth, lowest latency and smallest size, to L3. The hierarchy may be *exclusive*, so that an address is stored exactly once in any cache in the hierarchy, or *inclusive* so that the content of each cache is replicated at every level below it. An individual cache may be specialised for instructions or data, or may store both, and it may be shared amongst all the cores in a package, or between a subset of those cores, or it may be for the exclusive use of a single core. There are usually two L1 caches per core, one for instructions and one for data, with higher levels of cache being shared, and storing both instructions and data.

A substantial performance advantage exists for codes which arrange for all working data to already be in the processors' caches when it is required, but this may not always be possible. When it is not, and *cache misses* are common, then the serial execution speed will be dominated by the number of memory accesses, rather than the number of processor operations as has traditionally been assumed. Note also that when making scattered reads of a few bytes from many locations, all accesses must still be made in units of a cache line, and the vast majority of the data returned from main memory will be unused.

1.2.3 Cores

A core can be thought of as reading and executing a stream of instructions from main memory in the order in which they are stored. Operations within the core are synchronised by a clock pulse, and by raising the frequency of this clock, a core can be made to execute instructions more rapidly. However, the power consumption of a core is an approximately quadratic function of its clock frequency, with most of this energy being dissipated as heat.

Each instruction will require some number of clock cycles for the core to complete. Rather than increasing the clock frequency to improve performance, it is possible to instead reduce the number of instructions which are required, or to reduce the number of cycles per instruction. In the former case, arithmetic operations can be *vectorized*, so that fixed length arrays, or pairs of arrays, are modified by a single instruction.

Many instructions also involve common steps, so that by separating out the

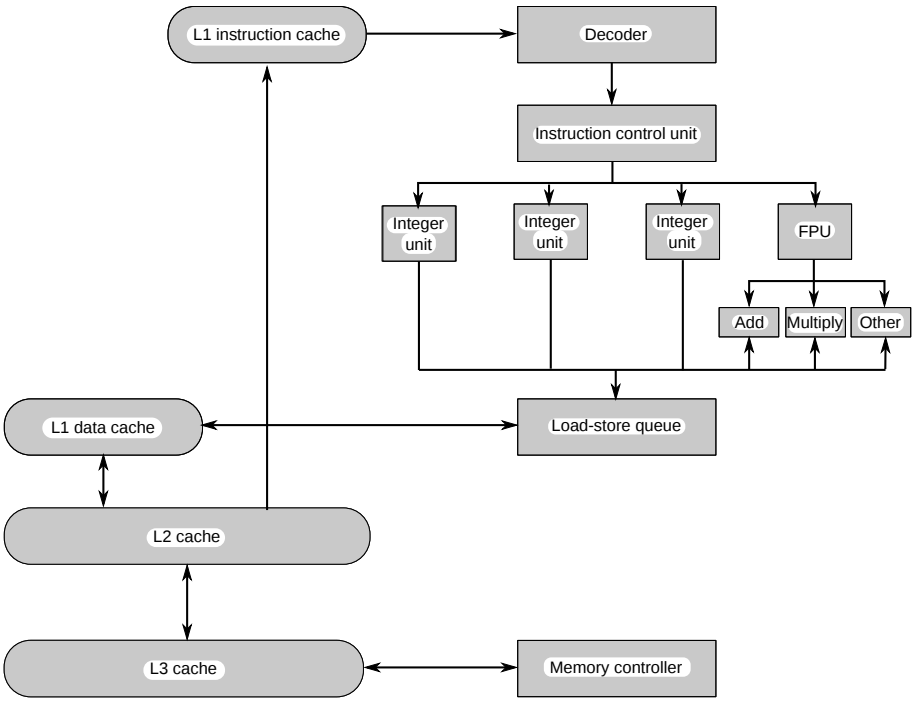


Figure 1.2: A simplified schematic of interactions in a typical processor core, based loosely on the AMD K10 architecture.

functional units involved in each stage, it is possible to perform different parts of several instructions simultaneously. This optimization, termed *pipelining*, significantly increases the rate at which instructions can be completed, even though many cycles are still required for each one. The individual stages are called *micro-operations* (μops).

In a *superscalar core*, two or more mutually independent instructions can be performed at once, as the core contains several units capable of performing each stage. To optimize the internal scheduling of micro-operations, the instruction stream can be re-ordered dynamically as it is read to bring more functional units into use. This is termed *out-of-order execution*, and is done is such a way as to be indistinguishable from executing the instructions in the order in which they were written.

Both pipelining and superscalar design require the instructions which will be executed next to be known, but this may be impossible due to conditional logic. The *branch prediction unit* guesses whether or not a branch will be taken. When it guesses incorrectly, all work performed on instructions which lie on the wrong side of the branch must be thrown away, which can be very expensive. Similarly, to overcome the latency of reads from main memory, a *predictive preloader* looks for patterns in the memory accesses a code performs, and loads the data which will be needed next into cache.

Branch mispredictions in tight loops can be avoided so long as one direction of branching is much more likely than the other; when the two sides are equally likely, then poor performance will result. A typical base rule for branch prediction is “forward not taken, backward taken” which can be used to optimize

code. Also, conditional stores can eliminate branching entirely for simple tests. To optimize for preloading, data should be accessed sequentially with a fixed stride.

A *load-to-store dependency* occurs when one instruction depends on the result of the previous instruction. In this case, the internal parallelism of the core will not be invoked, and many of the redundant superscalar units will sit idle. By spacing such instructions out, for example by performing several calculations at the same time, such waste can be avoided.

In *simultaneous multi-threading* (SMT), storage for several sets of registers and processor states is available for each core, giving multiple logical processors. Such *hardware threads* must all compete for the same execution resources and caches, however, and this can be to the detriment of high performance codes.

Sparse arithmetic, where many elements of each vector are zero, and efficiency requires operating only with the scattered nonzero elements, does not fit the previously described optimization guidelines. Sparse routines must be expected to achieve far lower instruction throughput than their dense counterparts, even though they remain more efficient than applying dense routines to mostly zero vectors. More information can be obtained from the optimization reference manuals provided by processor manufacturers [2, 1].

1.2.4 Packages

Several cores may be placed together upon a single silicon die, forming a *processor package*. Such cores are not fully independent, as they contend for the use of memory controllers and the larger memory caches. In addition, the package may change the clock frequency of its cores depending upon how many of them are in use, so that the speed of a core depends on the number of its peers which are active.

Multiple processor packages may be installed in a single system. Each package will have its own memory attached, so that main memory is fragmented. Reading data from another package's memory may be more expensive for a core than reading from its own. This difference is called *non-uniform memory access* (NUMA).

The computer system as a whole provides the illusion of a single memory space, but the contents of any location depend not just upon the associated DRAM, but also upon the caches of all the cores in all of the installed packages. To write to memory, a core must claim the memory addresses underlying the cache line of interest, invalidating any copies of those same addresses in all other cores' caches. This forms the basis of the MESI* *cache coherency protocol*.

These behaviours have several implications for software design. When working data is stored in the memory associated with a single package, the bandwidth to the memory of the other packages is unused, and additional latency is experienced by their cores. Similarly, writing to adjacent memory locations from different cores forces rapid changes of ownership of the relevant cache line, called *cache pinging*, which degrades performance.

The operating system has ultimate control over the physical location of data and processes, and this may exert an unfortunate influence on the performance

*MESI is named for the states each cache line may be in: modified, exclusive, shared or invalid.

of a parallel program. In particular, the operating system may favour scheduling to a single package for power performance, which can in turn lead to all active cores operating at lower clock frequencies than they might otherwise. In the presence of SMT, thread pairs may be favoured for scheduling together, even though these represent just a single set of execution resources.

1.2.5 GPUs

A graphics processing unit (GPU) is a separate device, including its own memory, which functions as a coprocessor. Its design is intended to permit calculations to be performed across a two-dimensional grid at high speed. This is made possible by assuming limited interdependence of the solution processes at different positions.

A typical device contains a set of *streaming multi-processors*, each of which is capable of processing simultaneously, in parallel, every element from a two-dimensional *block* of the calculation grid.* In turn, each block is subdivided into *warps*, and within a warp all computation is vectorized, and so occurs in lock-step. A single point in the grid is called a *thread*. If two threads in the same warp require different instructions to be executed, a *warp divergence* occurs, and both sets of instructions are performed, with the results for some threads being discarded. This is called *symmetric multi-threading*

Memory reads are also vectorized, which is called *memory coalescing*, so that the requests made by the threads in a warp are aggregated into a single block, and this allows a wider data bus to be used. When the threads in a warp do not simultaneously access adjacent memory locations, several memory requests must be issued, and this carries significant performance penalties.

Each streaming multi-processor also sets aside a small amount of local memory as data cache. This memory is arranged into *banks*, and each bank can service only one request at a time. When two threads access the same bank, a *bank conflict* occurs, and the accesses must be performed one after another.

Programs for the GPU are written as *kernels*. These are functions which execute on every thread in the grid, and where it is possible to communicate with other threads in the same warp without synchronization, and with other threads in the same block with synchronization. A kernel may execute on the threads in two different blocks in any order, so that these threads need not all be simultaneously active, and this makes synchronization between threads in different blocks impossible. Complex programs must therefore be broken up into many kernels, called in a strict sequence, so that each kernel can finish completely before the next begins.

1.2.6 Test systems

The results provided in this thesis were obtained from two test systems, which are described here for future reference.

`grunty` This machine has two quad-core AMD Opteron 2378 processor packages, each with a clock frequency of 2.4 GHz, and a total of 16 GiB of DDR2

*The terminology in this section is derived from NVIDIA's technical documentation. Other manufacturers may use different terms, but the concepts are much the same.

667 MHz SDRAM. The Opteron 2378 is based on AMD's Shanghai core, part of the K10 architecture, and fabricated at 45nm. It has separate 64 KiB 2-way set associative L1 caches for instructions and data for each core. Each core also has its own unified 512 KiB sixteen-way set associative L2 cache. The 6 MiB of unified L3 cache are shared between all cores in a package. The two packages are connected by AMD's HyperTransport (HT). The best supported vectorization on this platform is Streaming SIMD Extensions 2 (SSE2) and SSE4a.

`grunty` also has an NVIDIA Tesla C2070 graphics card with 6 GiB of 3 GHz GDDR5 RAM. The Tesla C2070 is based on NVIDIA's Fermi architecture. It contains 14 streaming multiprocessors, each clocked at 1.15 GHz, and is capable of executing 448 threads simultaneously. This gives the card an overall peak theoretical performance of 515.2 GFLOPS in double precision.

`richtmyer` This machine has two octo-core Intel Xeon E5-2670 processor packages clocked at a nominal 2.6 GHz, and a total of 64 GiB of 1600 MHz dual-ranked RDIMMs. The Xeon E5-2670 is based on Intel's Sandy Bridge architecture, and is fabricated at 32nm. Each core has separate 32 KiB eight-way set associative L1 caches for instructions and data, and a 256 KiB eight-way set associative L2 cache. The 20 MiB of L3 cache is shared amongst all the cores in a package. The two packages are connected by Intel's QuickPath Interconnect (QPI). This platform supports Advanced Vector Extensions (AVX), and contains a turbo-core facility: two cores may run at 3.3 GHz, four at 3.2 GHz, six at 3.1 GHz or eight at 3.0 GHz. Simultaneous multi-threading (Hyperthreading) is turned off for this machine.

1.3 Linear programming

The constrained mathematical programming problem takes the form

$$\begin{array}{rll} \text{maximise} & f(x) & \\ \text{subject to} & g_i(x) \leq 0 & i = 1 \ldots m_1 \\ & h_j(x) = 0 & j = 1 \ldots m_2 \\ & x \in \mathcal{X}. & \end{array} \tag{1.1}$$

Solving general problems of this form is extremely expensive, but as more restrictions are placed upon the constituent parts, f, g, h and $\mathcal{X}$, so the efficiency of the solution techniques available improves.

When there are no equality constraints, the set $\mathcal{X}$ is convex, and when the functions $-f$ and g_i are convex and differentiable on $\mathcal{X}$, (1.1) simplifies to the convex programming problem,

$$\begin{array}{rll} \text{maximise} & f(x) & \\ \text{subject to} & g_i(x) \leq 0 & i = 1 \ldots m. \end{array} \tag{1.2}$$

For such problems, the Karush-Kuhn-Tucker (KKT) [81] conditions,

$$\begin{aligned} \nabla f(x) &= \sum_{i=1}^{m} y_i \nabla g_i(x) \\ g_i(x) &\leq 0 \qquad\qquad i = 1 \dots m \\ y &\geq 0 \\ y_i g_i(x) &= 0, \qquad\qquad i = 1 \dots m \end{aligned} \tag{1.3}$$

are necessarily satisfied at any optimal point by some multipliers y, provided that certain regularity conditions on the problem hold. There are several possible such regularity conditions, with the most common being (i) linearity of f and all g_i; and (ii) Slater's condition [108] which requires that there exist a strictly feasible solution (i.e. x such that all $g_i(x) < 0$).

The linear programming problem is a further specialisation* of the convex programming problem, and is the main focus of this thesis. It may be stated as

$$\begin{aligned} \text{maximise} \quad & c^\top x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \\ & x, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}. \end{aligned} \tag{1.4}$$

Note that there are several equivalent formulations of the linear programming problem, and in any given section the most convenient for the material at hand will be used.

The KKT conditions are necessarily satisfied at any optimal solution of this problem, and may be stated in this context as

$$A^\top y \geq c \tag{1.5a}$$
$$Ax \leq b \tag{1.5b}$$
$$x, y \geq 0 \tag{1.5c}$$
$$y^\top (Ax - b) = 0 \tag{1.5d}$$
$$x^\top (A^\top y - c) = 0. \tag{1.5e}$$

Although the obvious strategies for solving (1.4) focus on the variables x given in the problem definition, the inherent symmetry of the KKT system suggests it is equally possible to solve for y,

$$\begin{aligned} \text{minimise} \quad & b^\top y \\ \text{subject to} \quad & A^\top y \geq c \\ & y \geq 0 \\ & y, b \in \mathbb{R}^m, c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}. \end{aligned} \tag{1.6}$$

This is simply the partner problem in y whose KKT conditions are the same as those for the original problem in x, and is called the *dual* of that problem.

*The presence of equality constraints in a linear program is consistent with this definition, as a linear equality can be written as two linear inequalities of opposite sign, both of which are convex.

Consider any y satisfying (1.5a) and (1.5c), and any x satisfying (1.5b) and (1.5c), then

$$c^\top x \leq y^\top A x \leq b^\top y, \tag{1.7}$$

which is termed *weak duality*, and holds for primal-dual pairs of many problem classes. If x and y additionally satisfy (1.5d) and (1.5e), then

$$c^\top x = y^\top A x = y^\top b, \tag{1.8}$$

which is termed *strong duality*: the optimal objective values of (1.4) and (1.6) are the same. This is one condition of the *fundamental strong duality theorem* [44].

1.3.1 The simplex method

Simplex methods are algorithms for solving linear programming problems, although there are simplex-like approaches for more general classes of program [121, 6, 47]. To simplify the discussion of their basic operation, consider a problem in *simplex normal form*,

$$\begin{aligned} \text{maximise} \quad & z \\ \text{subject to} \quad & Ax + s = b \\ & c^\top x - z = 0 \\ & x, s \geq 0, \end{aligned} \tag{1.9}$$

where *slack variables* s and an objective variable z have been added to (1.4) to create a system of equations with non-negative variables.

Suppose that $b \geq 0$, so that $(x, s, z) = (0, b, 0)$ is a feasible solution to (1.9). If $c_q > 0$, then increasing x_q will also increase z: if the equations are still to be satisfied then $x_q = \theta$ requires $z = \theta c_q$, and $s_i = b - \theta a_{iq}$.

There are now two possibilities. If $s_i \geq 0$ for all positive values of θ, then (1.9) is *unbounded*, which is to say that no optimal solution exists to the problem as posed. This can be observed trivially: given any proposed upper bound u on the objective, a feasible solution with a superior objective is obtained for $\theta = (u + 1)/c_q$.

Alternatively, there exists some θ' such that for any $\theta > \theta'$, $s \not\geq 0$, and a maximum step of θ' can be made in x_q. The previous procedure can now be repeated, obtaining a sequence of solutions which are monotonically increasing in the objective variable z, and terminating* at some solution $(\bar{x}, \bar{s}, \bar{z})$, or with the conclusion that the problem is unbounded.

Proposition 1.3.1. *There exist problems such that the solution $(\bar{x}, \bar{s}, \bar{z})$ can never be optimal.*

Proof. Consider the problem in Figure 1.3,

$$\begin{aligned} \text{maximise} \quad & x_1 + x_2 \\ \text{subject to} \quad & 2x_1 - 4x_2 \leq 1 \\ & -4x_1 + 2x_2 \leq 1 \\ & x_1 + 4x_2 \leq 4 \\ & 4x_1 + x_2 \leq 4, \end{aligned} \tag{1.10}$$

*There are finitely many variables, so for this simple algorithm, termination is guaranteed.

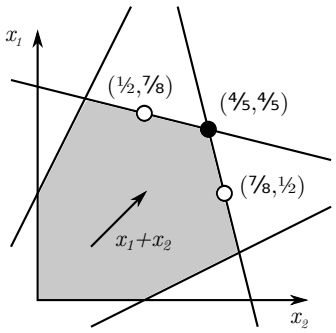


Figure 1.3: An example linear programming problem in two variables and four constraints.

which has optimal solution $(\frac{4}{5}, \frac{4}{5})$. First increasing x_1 gives solutions of $(\frac{1}{2}, 0)$, $(\frac{1}{2}, \frac{7}{8})$, while the case for x_2 is analogous. Note that if any element of b had been zero, a situation called *degeneracy*, then the previously described algorithm could make no progress toward a solution. □

Suppose that after increasing x_q, a slack s_p is identified which will become negative for $x_q > \theta'$. The goal is to reformulate the problem, so that the form (1.9) is restored, in which only slacks take nonzero values. Since $x_q \geq 0$, it must become the slack for some equation, and as $s_p = 0$, the p^{th} equation is a candidate. Solving for x_q in this equation yields

$$\sum_{j \neq q} \frac{a_{pj}}{a_{pq}} x_j + \frac{1}{a_{pq}} s_p + x_q = \frac{b_p}{a_{pq}}, \tag{1.11}$$

and replacing x_q throughout the remainder of the equations gives

$$\sum_{j \neq q} \left(a_{ij} - a_{iq} \frac{a_{pj}}{a_{pq}} \right) x_j - \left(a_{iq} \frac{1}{a_{pq}} \right) s_p + s_i = \left(b_i - a_{iq} \frac{b_p}{a_{pq}} \right) \qquad (i \neq p). \tag{1.12}$$

The z constraint can be treated similarly, yielding

$$\sum_{j \neq q} \left(c_j - c_q \frac{a_{pj}}{a_{pq}} \right) x_j - \left(c_q \frac{1}{a_{pq}} \right) s_p - z = \left(0 - c_q \frac{b_p}{a_{pq}} \right), \tag{1.13}$$

at which point the entire problem can be relabelled, by letting x'_q be s_p and s'_p be x_q, and a new problem obtained in the same form as (1.9),

$$\begin{aligned} \text{maximise} \quad & z \\ \text{subject to} \quad & A'x' + s' = b' \\ & {c'}^{\top} x' + z = b_0 \\ & x', s' \geq 0, \end{aligned} \tag{1.14}$$

for which $(0, b', b_0)$ is a feasible solution.

This modified step can now be applied repeatedly, terminating with a solution (x^*, s^*, z^*) or the conclusion that the problem is unbounded. The algorithm which results is the simplex method or, more precisely, the primal, column-wise simplex method.

Proposition 1.3.2. *If a solution* (x^*, s^*, z^*) *is found, it is an optimal solution to the problem* (1.9).

Proof. By reformulating the problem at each step, the simplex method has terminated with both a solution value and a problem statement of the form (1.14). To simplify notation, the primes in the reformulated problem will now be dropped; note that a solution in the original problem can simply be relabelled to become a solution in the reformulation, and vice versa.

There are two special properties of this reformulated problem:

i. $c \leq 0$: were there any component $c_q > 0$, further iterations would have occurred.

ii. $x^* = 0$: the point $(0, b, 0)$ was assumed to be an initially feasible solution, and at each iteration the component of x which increased became a slack in the subsequent iteration.

Consider any feasible point (x, s, z). Now $x \geq x^* = 0$, and $c \leq 0$, so

$$z = c^\top x + b_0 \leq c^\top x^* + b_0 = z^* \tag{1.15}$$

and hence z^* is optimal. □

Although Proposition 1.3.2 establishes that a solution returned by the simplex method is optimal, whether the simplex method does indeed terminate depends largely upon the means by which the variables to increase x_q are chosen. Criteria which lead to a method which provably terminates are not used in practice, because they require too many iterations to converge. See [18] for more details.

Geometrically, a linear programming problem has a feasible region bounded by the hyperplanes of the non-negativity constraints on the variables and slacks, and this means it is a polytope. The initial solution $(0, b, 0)$ is a vertex of this polytope, as at least n orthogonal hyperplanes intersect here in n dimensions (one for each variable). Both the simplex method and our original, greedy algorithm begin by moving along some axis until a constraint would become violated. This point is another vertex, as once again n linearly independent hyperplanes intersect in n dimensions.

The weakness of the greedy algorithm was that it could only move along the axes of the problem as stated. The simplex method, however, rotates the axes of the current problem so that they are the edges leading away from the current vertex, and can then follow these edges from vertex to vertex, up to an optimum. The rotation applied can be derived directly from the set of slacks, or *basis*, at a given iteration, which has particular significance in practice.

1.3.2 Simplex developments

The basic operation of the simplex method has now been described. A practical simplex solver, however, will contain numerous enhancements to improve numerical stability and performance. Some of the most important such modifications are now described.

The revised simplex method Take any iteration of the simplex method, and let B be a square matrix drawn from $[\,A\,I\,]$, consisting of those columns which multiply the variables currently labelled as slacks in the reformulated system. This problem can now be written

$$B^{-1}Ax + B^{-1}Is = B^{-1}b, \tag{1.16}$$

where the existence of an inverse follows from the operation of the simplex method. At each iteration, only the costs c in the updated problem are needed to select a variable to increase, and only the right-hand side b and column a_q of the current problem are needed to select a slack to relabel. Maintaining all of the mn coefficients in the formulation can thus be avoided if B^{-1} is maintained instead, and the resulting method is termed the *revised simplex method* [32], in contrast to the previously described *standard simplex method*.

General form Models are usually constructed in a more compact form than simplex normal form, for example

$$\begin{aligned} \text{maximise} \quad & c^\top x \\ \text{subject to} \quad & L \leq Ax \leq U \\ & \ell \leq x \leq u \\ & x, c \in \mathbb{R}^n, A \in \mathbb{R}^{m\times n}, \\ & \ell, u \in \mathcal{R}^n, L, U \in \mathcal{R}^m. \end{aligned} \tag{1.17}$$

where $\mathcal{R} \stackrel{\text{def}}{=} \mathbb{R} \cup \{-\infty, \infty\}$.

Rather than transforming such problems to simplex normal form, which requires additional variables and constraints, the more general form can be used directly in the simplex solver.

Feasibility The point $(0, b, 0)$ was previously assumed to be a feasible solution to the initial linear program, but there are valid programs with $b \not\geq 0$. Such problems can be treated by introducing new variables which satisfy the constraints, and whose activity is penalised by some large factor M. For sufficiently high penalties, this new program has the same solution set as the old if that program is feasible. In practice, solvers introduce such variables only implicitly, and M is treated symbolically rather than being assigned some particular value [91]. See Appendix A for more details.

Pricing The means by which the variable to increase x_q is selected at each iteration plays a major rôle in determining the number of simplex iterations required to solve a problem. An obvious strategy is to find the variable with the most attractive cost, but the iterations required to solve a problem can be reduced if c_q is scaled according to the steepness of the edge which it prices. This is called *normalised pricing*, and numerous methods for obtaining estimates of steepness exist, either from exact edge norms [48, 41], or an approximation to them [63, 28, 113, 41, 56] (see Appendix C for comparisons). To reduce instead the cost of each iteration, *partial pricing* may be employed, in which costs are available for only a subset of the columns at a given time. How to select an attractive subset of columns over which to optimize has been extensively studied (see e.g. [92] for a review).

Ratio test The ability to find a reformulation in which only slacks take nonzero values is dependent upon finding some s_p which becomes zero, and can thus be labelled a variable. This process is called the *ratio test*, and a numerically resilient version is based on the *thick pencil* approach [63], in which it is assumed to be acceptable to violate each constraint by some small amount - the "width" of the pencil mark. This additional freedom can allow a choice of slack to relabel, which is conventionally made for numerical stability.

At a degenerate vertex, the simplex method may be unable to make any progress before violating another slack. In the worst case this leads to non-termination, or *cycling*, but more typically performance is only degraded. To reduce these losses, a perturbation may be applied to b to remove the degeneracy [98, 122], or the feasible region may be continuously expanded to ensure progress is made [46].

The dual simplex method The simplex method so far described works directly on the primal problem, and is called the *primal simplex method*. The *dual simplex method* [84] is the same algorithm applied implicitly to the dual problem, whilst maintaining only the current primal formulation as before. The dual simplex method begins at a problem with $c \leq 0$, and at each iteration selects a slack to make feasible, terminating with $b \geq 0$. There is detailed discussion of a modern dual simplex implementation in [78].

1.3.3 Parallel simplex

For the standard simplex method, the calculation of the full reformulation of the problem (called the *tableau*) at each iteration has been parallelised on several generations of hardware [127, 36]. Performance is for the most part hardware limited, as the strictly serial parts consume little time, and near linear speed-up is possible.

For the revised simplex method, the calculation of πA, used to find reformulated costs, or rows of the tableau, has been parallelised many times [106, 66], and is hardware limited. Bixby and Martin [17] additionally performed the update of the reduced costs and the dual ratio test in parallel. If an explicit dense inverse of B is used, this can be updated with the same properties as the standard simplex tableau [106, 110].

Parallelising the remainder of an efficient revised simplex code is more difficult. Two asymmetric, task-parallel schemes were described by Hall and McKinnon [60, 61] which dedicate different parts of the simplex algorithm to different workers. PARSMI [60] has six distinct worker rôles and achieved speed-up of up to 3 on 6 processors. ASYNPLEX [61], with four worker rôles, achieved a reported speed-up of up to 5 on 13 cores. As this thesis was being written, Forrest [40] described ABOCA, a task-parallel division of the dual simplex method which uses Cilk [19] to provide self-balancing parallel primitives. The speed-up achieved reached 1.8 on two cores.

Exploiting problem structure to improve parallelism has been attempted by several authors. Boduroğlu [21] implemented a hybrid parallel standard simplex for problems with generalised upper bounds. Lubin et al. [86] implemented a revised method on a distributed memory cluster, making use of a structured basis for large, dual-angular, stochastic problems, and obtained some speed-ups of over 100x on 128 cores.

Structured problems can also be exploited with a decomposition strategy, so that several serial simplex solves occur in parallel [115, 65]. This has the disadvantage that the performance of such decomposition schemes is generally inferior to the revised simplex method in serial, even though their parallel performance is, for the most part, hardware limited [115].

A fuller review of the history of parallel linear programming in the context of the simplex method can be found in [59].

1.3.4 Interior point methods

Interior point methods are algorithms for solving general convex problems, including linear programming problems as a special case [128 translated in 75;70]. The underlying theory for interior point methods is somewhat involved, and beyond the scope of this thesis. As such, this section contains only a high level overview of the steps such methods perform. For a fuller description, including proofs of convergence and polynomial complexity, the interested reader is referred to [124, 103].

Consider a problem of the form

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \\ & x, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}, \end{aligned} \tag{1.18}$$

having dual

$$\begin{aligned} \text{maximise} \quad & b^\top y \\ \text{subject to} \quad & A^\top y + s = c \\ & s \geq 0 \\ & s, c \in \mathbb{R}^n, y, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}. \end{aligned} \tag{1.19}$$

Suppose feasible solutions are known for the primal problem, $\bar{x}$, and the dual problem, $(\bar{y}, \bar{s})$, and that these solutions lie in the *interior* of the feasible region, that is to say $(\bar{x}, \bar{y}, \bar{s}) > 0$. The original problem may be modified by adding a *barrier* term, as in

$$\begin{aligned} \text{minimise} \quad & c^\top x - \mu \sum_{i=1}^{n} \log x_i \\ \text{subject to} \quad & Ax = b \\ & x > 0, \end{aligned} \tag{1.20}$$

and this has dual

$$\begin{aligned} \text{maximise} \quad & b^\top y + \mu \sum_{i=1}^{n} \log s_i \\ \text{subject to} \quad & A^\top y + s = c \\ & s > 0. \end{aligned} \tag{1.21}$$

Although the objective function is no longer linear in the barrier problems, it is nevertheless convex, and Slater's condition (the existence of an interior point)

is satisfied by assumption. Therefore, at any optimal point the appropriate KKT conditions,

$$\begin{aligned} Ax - b &= 0 \\ A^\top y + s - c &= 0 \\ x_i s_i &= \mu \\ x, s &> 0, \end{aligned} \tag{1.22}$$

hold by necessity. Note that although the complementarity condition has been perturbed, the remainder of (1.22) are identical to the KKT conditions for the original problem.

An optimal solution to the original problem is not necessarily unique, rather it lies in some linear subspace of dimension $r \geq 0$. However, it is a surprising fact that (1.22) have a unique solution in the positive orthant for any $\mu > 0$. The curve defined by such solutions is called the *central path*.

Intuitively, rather than seeking one of the many solutions for which $\mu = 0$, the central path is pursued as $\mu \to 0$ in the modified problems, in the hope that this will reveal a solution for $\mu = 0$. Surprisingly, this is a successful strategy, and (x, y, s) on the central path tend to the analytic centre of the set of optimal solutions for the original problems as μ tends to zero.

For such methods to be practical, it is necessary to be able to solve for points on the central path to some degree of approximation. Dropping the positivity requirements on x and s, and moving μ to the left hand side, this can be considered to be a problem of the form

$$F(x, y, s) = 0, \tag{1.23}$$

so that a search for a solution to the perturbed KKT conditions can be recast as finding a root of F which is non-negative in x and s.

One general approach to finding the roots of functions is Newton's method [100]. To apply it to (1.22), the KKT equations are linearised around the current estimate at each iteration, and a search direction is found using

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^\top & I_n \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^\top y - s \\ \mu - XSe \end{bmatrix}, \tag{1.24}$$

which can be calculated directly by inverting the left hand side. A subsequent line search is performed to determine the distance of the step to make. It is surprising that this process will reliably converge to a positive root, given how few restrictions are placed upon the initial estimate.

1.3.5 Interior point method extensions

As for the simplex method, an interior point solver will contain many enhancements for speed, convergence and stability. This section briefly outlines some of the most important.

Predictor-corrector Rather than solve accurately for each point on the central path, it is sufficient to make an approximation to it, and then update from the new position. Mehrotra [93] first described the pattern used in many existing

solvers: the usual update is called the *predictor* and is taken as a first estimate of the step to make. A second solve is then performed to calculate the *corrector*, which balances distance from the central path with immediate progress in decreasing μ, and also contains second order terms from the predictor.

The normal equations The equations (1.24) are not in practice solved in this form. The most common reduction is to a symmetric, positive definite form now called the *normal equations* [96]

$$(AXS^{-1}A^\top)\Delta y = AXS^{-1}\left(A^\top y - c + \sigma\mu X^{-1}e\right) + Ax - b, \tag{1.25}$$

where σ is the centering parameter. The normal equations are discussed in detail in [5].

Cholesky factorisation The most expensive part of an interior point iteration is the factorisation of the normal equations. The standard technique is to use a Cholesky factorisation of the form $AXS^{-1}A^\top = LL^\top$ [14], in which pivoting for numerical stability is unnecessary. This means that an extensive symbolic analysis can be performed before the first iteration to identify structure in the target matrix, as this analysis will remain valid throughout the solve [67].

Infeasible methods The existence of a strictly interior point has so far been assumed, but there exist feasible linear programs for which such a point does not exist. Infeasible interior point methods [79] remove this restriction, and are thus preferred in practice. Although all iterates generated are strictly infeasible, they tend as before to an optimal solution.

1.3.6 Parallel interior point methods

Unlike the simplex method, interior point methods have been shown to lend themselves naturally to parallelisation. Firstly, significant speed-ups have been reported for parallel interior point solvers for structured problems [88, 26, 35, 54, 107], with speed-ups ranging from 6 times on 18 cores [88], to 22 times on 24 cores [52]. For unstructured problems, parallelising just the Cholesky factorisation can give excellent results [16, 71, 8]. Reported speed-ups range as high as 9 times on 16 cores [8] and 108 times on 256 cores [71].

Chapter 2

i6 and i8 - Parallel standard simplex

2.1 Introduction

The standard simplex method has been implemented for parallel hardware on numerous occasions [e.g. 72, 29, 36, 116, 126, 85, 82], with reported speed-up ranging from 50 times on 64 processors, to 12 times on 16 processors. These results are a consequence of the update of the tableau consuming the majority of such a code's runtime, and that update being *embarrassingly parallel* - the calculation of each coefficient can be performed independently.

However, the standard simplex method does not form the basis of competitive solvers for general problems. This may be attributed to three causes. Firstly, few of the tableau coefficients are required at each iteration, and maintaining them all is far more expensive than updating a sparse basis factorisation and calculating only the required elements.

Secondly, the tableau requires a dense memory block of the same dimensions as the constraint matrix, and for large problems this may be an unacceptably large amount of storage. In contrast, the sparse revised simplex method has memory requirements in line with the number of nonzeroes in the constraint matrix, and this is typically orders of magnitude less.

Thirdly, whilst a new factorisation of the basis matrix can be calculated efficiently, and at regular intervals, in the revised simplex method, rebuilding the standard simplex tableau is so expensive as to make it a last resort in any practical code. This leads to numerical error accumulating during a solve.

However, for dense, near-square, linear programming problems, the standard simplex method is competitive even in serial. Such problems can arise in practice as, for example, master programs in decomposition methods [72].

This chapter describes three novel implementations of the standard simplex method on parallel hardware. They are shown to have a substantial performance advantage over two commercial solvers on a class of random, dense problems.

2.2 Multi-core standard simplex

`i6` and `i6b` are novel implementations of the standard, primal simplex method for multi-core machines, which are capable of limited maximum improvement pricing. `i6` is optimized for general problems, which typically have sparse tableaux, whereas `i6b` is optimized for the fully dense case.

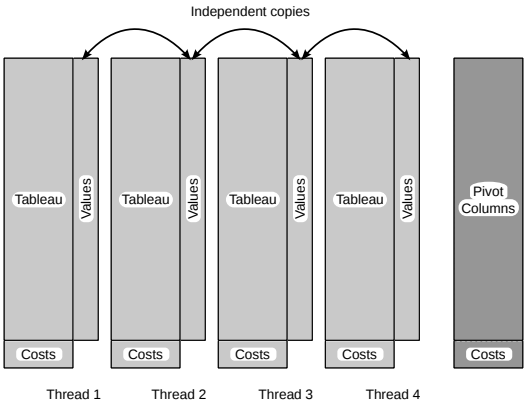


Figure 2.1: Data layout with four workers in `i6` and `i6b`. Communication is limited to placing a pivot column into the shared area at each iteration.

2.2.1 Algorithmic elements

Only the reduced tableau, $B^{-1}N$, is stored, where N is the matrix of the non-basic columns. Updates are performed uniformly column-wise, by first resetting the pivot column to the identity, and then applying the same update as for every other column. This requires that the pivot column be cached before the update begins.

In `i6`, the nonzero positions of the pivot column are packed into an indexed array to reduce the number of memory locations touched, and the update of any column with a zero in the pivot row is skipped. In `i6b`, these steps are not performed, so that a memory copy is sufficient, and vectorization of the update arithmetic can occur.

Both codes perform phase one and two simultaneously, using steepest-edge [80], but not composite [90], pricing. During the update, both the most attractive column for improving feasibility (phase one), and the most attractive column for improving the objective (phase two), are found, but a phase one candidate is always preferred. The steepest edge weights are updated trivially by

$$\|a_i + \Delta a_i\|^2 = \|a_i\|^2 + 2a_i^\top \Delta a_i + \|\Delta a_i\|^2. \tag{2.1}$$

Neither code will rebuild the tableau during a solve. To mitigate against numerical error, the update is performed without a drop tolerance, so that perturbed zeroes are not ignored. Also, extended precision is disabled in the floating point units, which ensures that the result of a calculation does not depend upon when a value is stored to memory.

The ratio test has two passes, the first being taken to find the largest step that can be made within slightly expanded bounds, the second to find a numerically stable pivot [63]. An infeasible basic variable may be moved to its far bound, but this method does not create new infeasibilities.

As discussed in § 1.2, all cores in a package compete for memory bandwidth and, on the test machines, a single core can consume the entire amount available for its package. To reduce memory traffic as far as possible, only a single sweep is required to update the tableau and edge norms, and to find the new incoming column. This modification is believed to be novel, and means that only the pivot column is read into memory more than once per iteration.

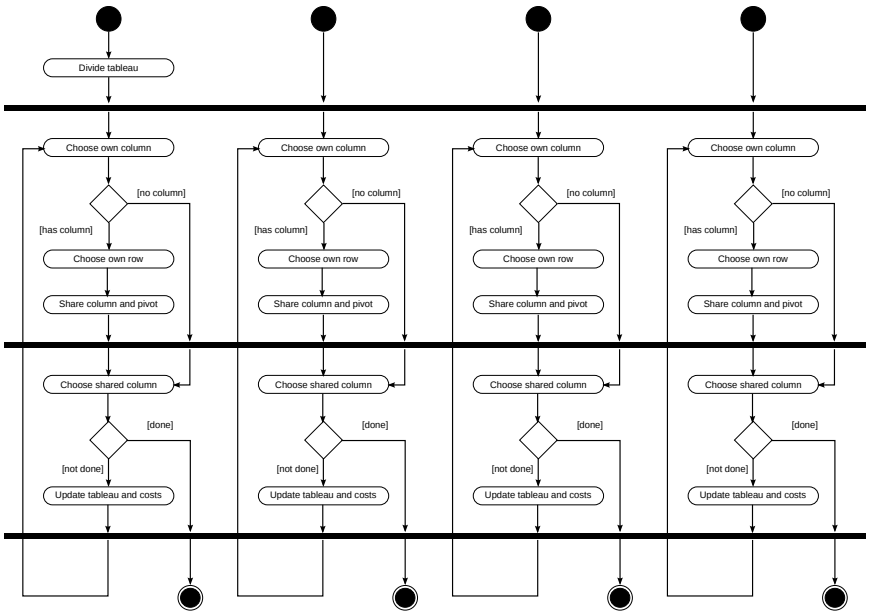


Figure 2.2: Task parallelism with four threads in `i6` and `i6b`. Synchronisation occurs twice per iteration, as the shared area moves from read- to write-accessible.

2.2.2 Parallelism

As shown in Figure 2.1, the tableau is divided column-wise into stripes for parallel solution, and each worker is assigned one stripe. The columns making up a particular stripe are drawn at random from the constraint matrix, which improves the load balance between workers on subsequent updates.

To avoid cache pinging, and improve data locality, one worker never directly accesses the data of another, instead an area is set aside for the exchange of candidates for the next pivot. At every iteration, each worker finds its most promising incoming variable, performs a ratio test, and then stores the results into the shared area. Each worker separately maintains its own copy of the entire primal solution, but need only store reduced costs for its own columns. This is shown in Figure 2.2.

Synchronising the workers after they have performed a ratio test on their own candidate column is believed to be a novel approach. Previous column-wise tableau implementations [e.g. 126] shared candidate columns first, and only then performed a ratio test. One advantage of this scheme is the ability to perform limited maximum improvement pricing between the candidates, where the best improvement in the objective can be selected at no additional cost.

However, using maximum improvement pricing is not the default behaviour of these codes, as despite its obvious attractions, this pricing scheme does not appear to be superior to steepest edge normalisation (see Appendix C). Also, `i6` and `i6b` without this technique are not only deterministic, but also follow identical paths with any number of threads, which is desirable in practice.

2.3 Many-core standard simplex

`i8` is an implementation of the standard, primal simplex method for GPUs. It is written for NVIDIA devices, and consists in main part of sixteen CUDA kernels. This is a complete simplex code, with all subroutines implemented entirely on the device, which is able to begin its solve from an infeasible point, select pivots in a numerically stable fashion, and make use of normalised pricing.

Previous implementations of the simplex method for such devices [55, 110, 82, 85, 15] have not been shown to be capable of solving real-world problems, have not been tested on reproducible problem sets, and have been not been benchmarked against competitive codes. As such, `i8` is believed to be the first practical simplex code for a GPU.

2.3.1 Algorithmic elements

A critical limit on the performance of GPU codes is the amount of data which must be transferred between the host and the device [3]. In `i8`, all of the major operations of the standard simplex method are performed in kernels, so that only individual scalars and indices are read from the device at each iteration. The host coördinates the solve, but is otherwise unused.

The reduced tableau $B^{-1}N$ is stored, and updated, column-wise on the device, and to reduce the number of kernels which need to be scheduled, the update is non-uniform, with the pivot column handled specially. The tableau is treated as entirely dense. Both the pivot column and pivot row are cached before the update begins, and to remove conditions from the update of a column, the pivot row is first destroyed and then corrected using the cached copy.

As for the multi-core solvers, pricing uses steepest edge normalisation [80], but the weights are recalculated directly during each tableau update. This is a pure phase one code: columns are selected to improve feasibility, until there are no such columns, and from then on columns are selected to improve the objective function. The ratio test has two phases [63], but can only step a variable to its near bound.

2.3.2 Parallelism

The tableau update is performed with one warp (32 threads) per column, with every 32^{nd} row being assigned to a given thread, ensuring memory access is coalesced. Steepest edge subnorms are accumulated in shared memory, and aggregated in parallel with $\log_2(32) = 5$ additions. Phase one and two costs are updated by a second kernel. Explicit synchronisation is unnecessary throughout.

To choose a column on the device, a work array is first filled with the normalised costs of all attractive variables. A kernel reduces this array to its maximum element in stages, as follows. Each block performs a complete reduction of a section of the array, using barriers for synchronisation. Repeated calls to the kernel are used to ensure that all blocks have completed their reduction before the next stage begins.

To choose a row on the device, six kernels are used. The first tests all rows to find the maximum step which preserves their feasibility with slightly expanded bounds, saving this to a work array. The second performs a parallel reduction to find the minimum such step. The third compares the column's step within

its expanded bounds to this minimum. The fourth finds each potential pivot and writes it to the work array. The fifth performs a parallel reduction to find the largest pivot. Finally, the sixth evaluates the column bounds, and signals a bound swap if this is attractive.

The remainder of the sixteen kernels in `i8` initialise steepest edge weights, correct phase one costs for feasibility changes, interchange leaving and entering variables, and update the primal solution.

2.4 Results

This section first presents results for real-world problems of small size, for which the standard simplex method is viable, but not generally competitive. This is to demonstrate that these codes are not limited to artificial instances as many previous attempts have been. Results are then provided for a class of random, dense problems for which the standard simplex method is expected to be more attractive, and show these codes to have a significant performance advantage over two commercial solvers.

In the following tables, a solution time is given for each code, along with an accuracy score α, which is calculated as

$$\alpha = \left\lfloor -\log_{10} \left| \frac{z - z^*}{z^*} \right| \right\rfloor, \tag{2.2}$$

where z is the objective the solver obtains, and z^* is the best known solution. This is a measure approximately equivalent to the number of accurate significant figures the solver achieves. The version of `cplex` used to generate these results was 12.3. The version of `gurobi` used was 4.5.1. Note that the results given below are from a single sweep, and that no problem specific tuning is applied to any solver.

2.4.1 Standard test problems

The results in this section are for Netlib [99], which is a standard test set for linear programming. These are no longer considered to be large problems, but as a collection which have historically caused difficulties for solvers, they are a meaningful test of a code's stability and resilience. A few problems are known to be very hard, for example `PILOT`, `PILOT87` and `DFL001`.

`i6` successfully solves 90% of the Netlib set to $\alpha \geq 5$. The failing problems may be divided into three groups. The problems `GROW22`, `NESM`, `PEROLD`, `PILOT.JA`, `PILOT4`, `PILOTNOV` and `TRUSS` are simply solved very inaccurately, which may be attributed to accumulated numerical error during the solve. The problems `QAP12`, and `QAP15` cannot be solved within the allowed time of an hour. The remaining two problems, `PILOT` and `PILOT87`, cannot be solved due to numerical failure. Results for `i6b` are similar, but `MAROS-R7` is additionally solved with low accuracy.

`i8` successfully solves 95% of the Netlib set to $\alpha \geq 5$. As before, the failing problems fall into three groups. The problems `GROW22` and `PILOT87` are solved to low accuracy. The problem `MAROS-R7` is unsolved due to numerical failure. Solves for the problems `QAP12` and `QAP15` exceed the maximum allowed time of one hour.

The commercial, revised simplex codes are significantly superior for these problems, solving all but a handful of instances in negligible time, and all of the remainder in fifteen minutes or less.

`i6` has an 88% performance advantage in serial over `i6b`, and a 75% performance advantage in parallel. Speed-up for both solvers peaks at 12 on 8 cores for `MAROS`, with `i6` showing super-linear speed-up for a further 10 problems. This can be explained by the additional cache which becomes available as more cores are committed. For the remaining 83 problems whose solve completes, the geometric mean of the speed-up for `i6` is 2.37 on 8 cores. This is consistent with speed being memory limited: eight cores have approximately double the available bandwidth of one, as they are distributed across two physical packages, each with its own dedicated memory.

2.4.2 Dense test problems

The results in this section are for a class of random, dense problems having the form

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & -ne \leq Ax \leq ne \\ & x \geq 0 \\ \text{where} \quad & a_{ij} \sim \mathrm{U}(-1, 1) \\ & c \sim \mathrm{U}(-1, 0) \\ & c \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}. \end{aligned} \tag{2.3}$$

The instances generated are difficult, in the sense that all tested codes require a large multiple of n iterations to solve them.

All tested problems can be solved accurately, but results are shown for only one of the largest instances, as the commercial solvers take significant time to reach optimality. Results for the dual simplex code of `gurobi` are also included, to demonstrate that these problems appear to favour solution by the primal simplex method.

In serial, the standard simplex codes outperform the revised solvers on every instance, and this advantage only improves in parallel. For the largest problem, a $10,000 \times 10,000$ dense matrix, the solution time for `cplex` primal simplex is approximately 25 days. The fastest revised code tested is the primal simplex of `gurobi`, at around 3 days. The fastest standard simplex code, `i8`, requires approximately 40 minutes. Even for the multi-core codes, solution time is under six hours - over ten times faster than the fastest revised solver.

On the smallest problems, speed-up for `i6` ranges between 7.6 and 9.0 on 8 cores, dropping to 2.6 on the largest problem. Speed-up for `i6b` ranges between 9.8 and 12.8 on the smallest problems, dropping to 2.3 on the largest. `i6b` also experiences an approximately 8% performance advantage over `i6` in serial, rising to 14% in parallel.

Problem			i6 (1)		i6 (8)		i6b (1)		i6b (8)		i8		gurobi (P)	cplex (P)
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)
25FV47	821	1571	9.78	11	1.11	11	11.28	11	1.14	11	4.22	11	0.52	0.29
80BAU3B	2262	9799	152.30	11	55.32	11	342.90	11	97.27	11	29.98	11	0.36	0.26
ADLITTLE	56	97	0.00	10	0.22	10	0.00	10	0.09	10	0.05	10	0.00	0.00
AFIRO	27	32	0.00	11	0.10	11	0.00	11	0.07	11	0.01	11	0.00	0.00
AGG	488	163	0.01	10	0.05	10	0.02	10	0.07	10	0.11	10	0.01	0.01
AGG2	516	302	0.03	11	0.07	11	0.06	11	0.10	11	0.19	11	0.01	0.01
AGG3	516	302	0.03	11	0.06	11	0.06	11	0.15	11	0.19	11	0.01	0.01
BANDM	305	472	0.21	12	0.15	12	0.23	12	0.20	12	0.90	12	0.05	0.03
BEACONFD	173	262	0.01	11	0.09	11	0.01	11	0.18	11	0.11	11	0.00	0.00
BLEND	74	83	0.00	11	0.09	11	0.00	11	0.15	11	0.07	11	0.00	0.00
BNL1	643	1175	1.42	8	0.42	8	1.79	10	0.35	10	1.55	11	0.16	0.13
BNL2	2324	3489	76.47	10	28.79	10	87.74	10	25.19	10	12.52	10	0.95	0.32
BOEING1	351	384	0.13	11	0.17	11	0.18	11	0.22	11	0.63	11	0.01	0.02
BOEING2	166	143	0.01	10	0.12	10	0.01	10	0.12	10	0.15	10	0.00	0.01
BORE3D	233	315	0.02	11	0.12	11	0.03	11	0.05	11	0.17	11	0.01	0.00
BRANDY	220	249	0.04	11	0.16	11	0.07	11	0.11	11	0.41	11	0.02	0.02
CAPRI	271	353	0.03	8	0.15	8	0.11	8	0.10	8	0.39	10	0.02	0.02
CYCLE	1903	2857	11.38	11	3.67	11	28.31	11	9.04	11	3.41	11	0.19	0.08
CZPROB	929	3523	6.06	10	2.85	10	17.17	10	6.11	10	3.24	10	0.10	0.06
D2Q06C	2171	5167	289.73	6	110.57	6	309.12	6	139.91	6	28.91	6	2.87	1.78
D6CUBE	415	6184	142.29	5	42.68	5	130.58	5	43.31	5	19.40	10	1.24	2.35
DEGEN2	444	534	0.28	12	0.27	12	0.38	12	0.35	12	0.96	14	0.12	0.13
DEGEN3	1503	1818	20.61	7	6.73	7	25.60	7	9.78	7	6.68	14	0.71	0.74
DFL001	6071	12230			2550.70	5			2876.63	5	752.97	9	45.15	21.39
E226	223	282	0.04	9	0.23	9	0.05	9	0.15	9	0.30	9	0.02	0.01
ETAMACRO	400	688	0.16	8	0.13	8	0.33	8	0.24	8	0.74	8	0.04	0.01
FFFFF800	524	854	0.35	11	0.22	11	0.60	11	0.28	11	0.81	11	0.01	0.03
FINNIS	497	614	0.09	6	0.29	6	0.28	6	0.15	6	0.64	6	0.01	0.02
FIT1D	24	1026	0.15	11	0.20	11	0.13	11	0.11	11	1.16	11	0.09	0.05
FIT1P	627	1677	5.53	11	0.68	11	5.51	11	0.69	11	1.70	11	0.06	0.11
FIT2D	25	10500	11.85	11	1.50	11	10.77	11	1.42	11	8.93	11	0.72	0.50
FIT2P	3000	13525	1821.77	9	593.15	9	1712.42	10	671.37	10	143.34	9	2.10	2.92
FORPLAN	161	421	0.22	11	0.29	11	0.23	11	0.16	11	0.55	11	0.02	0.02

Netlib results on ***grunty*** *from § 2.4.1. Continued on the next page.*

Problem			i6 (1)		i6 (8)		i6b (1)		i6b (8)		i8		gurobi (P)	cplex (P)
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)
GANGES	1309	1681	1.29	11	0.34	11	11.47	11	2.31	11	3.49	11	0.07	0.03
GFRD-PNC	616	1092	0.20	11	0.14	11	1.14	11	0.28	11	1.13	11	0.03	0.02
GREENBEA	2392	5405	301.48	11	94.34	11	374.58	11	159.60	11	38.16	12	1.54	0.60
GREENBEB	2392	5405	194.88	11	85.22	11	261.05	11	119.00	11	27.66	11	0.98	0.48
GROW7	140	301	0.01	11	0.24	11	0.03	11	0.18	11	0.20	11	0.01	0.04
GROW15	300	645	0.26	10	0.25	10	0.36	11	0.23	10	0.74	8	0.05	0.09
GROW22	440	946	1.57	0	0.44	0	1.39	0	0.43	0	1.49	2	0.12	0.11
ISRAEL	174	142	0.01	11	0.19	11	0.01	11	0.25	11	0.12	11	0.01	0.01
KB2	43	41	0.00	11	0.11	11	0.00	11	0.08	11	0.04	11	0.00	0.00
LOTFI	153	308	0.03	11	0.12	11	0.02	11	0.06	11	0.15	11	0.00	0.01
MAROS	846	1443	5.39	10	0.52	10	7.61	10	0.73	10	2.97	9	0.20	0.11
MAROS-R7	3136	9408	261.86	9	87.26	9	514.23	4	184.06	4	585.78		1.14	2.81
MODSZK1	687	1620	3.58	10	0.69	10	3.78	9	0.45	9	1.34	9	0.08	0.07
NESM	662	2923	12.18	10	1.41	10	16.68	10	1.78	10	3.82	10	0.42	0.19
PEROLD	625	1376	8.31	0	1.06	0	7.77	0	1.03	0	4.89	8	0.32	0.19
PILOT	1441	3652	9.63		3.38		141.26	0	54.12	0	33.74	5	3.04	2.33
PILOT.JA	940	1988	17.11	0	1.87	0	17.68	0	1.86	0	7.53	6	1.00	0.57
PILOT.WE	722	2789	17.49	9	2.81	9	20.44	9	2.81	9	6.04	10	0.34	0.32
PILOT4	410	1000	0.96	0	0.34	0	1.05	0	0.41	0	2.97	7	0.12	0.11
PILOT87	2030	4883	304.87		105.20		531.40		175.89		168.67	1	8.50	7.52
PILOTNOV	975	2172	15.27	0	2.88	0	14.93	0	2.74	0	4.66	11	0.26	0.32
QAP8	912	1632	18.20	7	1.97	7	17.67	7	1.64	7	4.47	13	0.87	0.78
QAP12	3192	8856											69.20	55.71
QAP15	6330	22275											778.64	771.00
RECIPE	91	180	0.00	16	0.19	16	0.00	16	0.18	16	0.04	16	0.00	0.00
SC105	105	103	0.00	11	0.08	11	0.00	11	0.07	11	0.06	11	0.00	0.00
SC205	205	203	0.01	11	0.05	11	0.02	11	0.08	11	0.15	11	0.01	0.01
SC50A	50	48	0.00	11	0.09	11	0.00	11	0.10	11	0.03	11	0.00	0.00
SC50B	50	48	0.00	15	0.06	15	0.00	15	0.07	15	0.03	15	0.00	0.00
SCAGR7	129	140	0.01	10	0.05	10	0.01	10	0.15	10	0.12	10	0.00	0.00
SCAGR25	471	500	0.25	10	0.19	10	0.35	10	0.13	10	0.74	10	0.01	0.03
SCFXM1	330	457	0.07	10	0.22	10	0.14	10	0.16	10	0.49	10	0.01	0.02
SCFXM2	660	914	0.59	13	0.26	13	1.18	13	0.39	13	1.32	13	0.05	0.05

Netlib results on **grunty** *from § 2.4.1. Continued on the next page.*

Problem			i6 (1)		i6 (8)		i6b (1)		i6b (8)		i8		gurobi (P)	cplex (P)
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)
SCFXM3	990	1371	3.12	11	0.53	11	6.63	11	0.79	11	2.52	11	0.04	0.08
SCORPION	388	358	0.03	10	0.12	10	0.11	10	0.22	10	0.44	10	0.00	0.00
SCRS8	490	1169	0.59	12	0.18	12	0.92	12	0.30	12	1.03	12	0.05	0.03
SCSD1	77	760	0.02	11	0.10	11	0.02	11	0.20	11	0.13	11	0.01	0.01
SCSD6	147	1350	0.18	8	0.15	8	0.17	8	0.11	8	0.39	10	0.03	0.03
SCSD8	397	2750	3.33	9	0.37	9	4.28	11	0.50	11	1.55	11	0.05	0.10
SCTAP1	300	480	0.04	15	0.21	15	0.10	15	0.15	15	0.33	15	0.01	0.01
SCTAP2	1090	1880	0.95	10	0.23	10	5.72	10	0.97	10	1.65	10	0.01	0.03
SCTAP3	1480	2480	1.98	14	0.77	14	13.72	16	5.75	16	2.91	15	0.02	0.05
SEBA	515	1028	0.76	15	0.26	15	1.12	15	0.41	15	0.86	14	0.00	0.01
SHARE1B	117	225	0.01	10	0.11	10	0.01	10	0.14	10	0.23	11	0.01	0.01
SHARE2B	96	79	0.00	11	0.07	11	0.00	11	0.06	11	0.09	11	0.00	0.00
SHELL	536	1775	0.24	16	0.11	16	2.46	16	0.32	16	0.99	16	0.02	0.02
SHIP04L	402	2118	0.16	10	0.15	10	1.09	10	0.36	10	0.50	10	0.01	0.02
SHIP04S	402	1458	0.13	10	0.12	10	0.52	10	0.25	10	0.48	10	0.01	0.01
SHIP08L	778	4283	0.47	11	0.43	11	7.55	11	3.14	11	1.24	11	0.02	0.04
SHIP08S	778	2387	0.24	10	0.19	10	3.93	10	0.60	10	1.07	10	0.01	0.01
SHIP12L	1151	5427	1.50	10	0.34	10	22.17	10	10.72	10	2.89	10	0.05	0.05
SHIP12S	1151	2763	0.48	11	0.28	11	10.48	11	3.89	11	2.15	11	0.02	0.02
SIERRA	1227	2036	0.54	10	0.20	10	7.11	10	3.45	10	1.74	10	0.02	0.03
STAIR	356	467	0.25	10	0.20	10	0.25	10	0.19	10	0.70	10	0.03	0.07
STANDATA	359	1075	0.01	15	0.21	15	0.06	15	0.20	15	0.09	15	0.01	0.01
STANDGUB	361	1184	0.01	15	0.10	15	0.07	15	0.14	15	0.09	15	0.00	0.01
STANDMPS	467	1075	0.08	15	0.16	15	0.42	15	0.25	15	0.45	16	0.00	0.01
STOCFOR1	117	111	0.00	10	0.13	10	0.00	10	0.19	10	0.06	10	0.00	0.00
STOCFOR2	2157	2031	17.43	11	8.34	11	24.54	11	10.18	11	4.74	11	0.08	0.09
STOCFOR3	16675	15695			2855.63	11			3397.68	11	900.22	11	3.34	0.90
TRUSS	1000	8806	189.18	10	71.31	10	195.89	10	71.66	10	19.49	11	1.20	0.81
TUFF	333	587	1.82	1	0.54	1	0.98	2	0.30	2	0.72	10	0.03	0.02
VTP.BASE	198	203	0.03	10	0.22	10	0.03	10	0.09	10	0.36	10	0.01	0.00
WOOD1P	244	2594	0.84	10	0.20	10	1.14	10	0.26	10	0.76	10	0.16	0.04
WOODW	1098	8405	60.91	8	23.44	8	78.00	11	34.61	11	8.29	10	0.15	0.04

Netlib results on ***grunty*** *from § 2.4.1.*

Problem			Solution time (s)						
Rows	Columns	Seed	`gurobi` (P)	`gurobi` (D)	`cplex` (P)	`i6` (1)	`i6` (8)	`i6b` (1)	`i6b` (8)	`i8`
1000	1000	1095	55.29	72.16	35.06	17.90	2.48	17.20	1.45	6.06
1000	1000	1223	51.66	63.98	31.84	13.62	1.65	12.54	1.31	5.61
1000	1000	1241	56.59	63.65	35.53	13.74	1.59	12.73	1.36	5.67
1000	1000	1514	55.70	68.86	32.89	13.36	1.53	12.30	1.26	5.50
1000	1000	2151	56.41	64.05	31.35	14.01	1.63	13.03	1.34	5.78
1000	1000	0295	60.13	66.37	33.40	13.79	1.56	12.66	1.40	5.65
1000	1000	3252	58.11	71.31	35.95	14.02	1.65	13.00	1.35	5.75
1000	1000	0359	55.69	67.37	32.51	13.82	1.62	12.78	1.35	5.71
1000	1000	5152	57.10	58.95	33.20	12.85	1.52	11.91	1.29	5.29
1000	1000	9952	53.52	68.69	33.35	14.51	1.69	13.40	1.42	5.97
3000	3000	1095	2990.76	3039.36	1563.49	693.37	164.70	645.92	163.04	59.32
3000	3000	1223	2617.50	3032.65	1469.52	669.67	227.04	623.71	191.96	57.27
3000	3000	1241	2849.53	2917.85	1513.31	674.93	264.47	629.14	250.78	57.75
3000	3000	1514	2812.67	2940.89	1483.32	673.76	228.46	627.61	234.86	57.08
3000	3000	2151	3194.38	3441.68	1644.65	681.56	183.52	634.58	193.00	58.22
3000	3000	0295	2819.41	2952.58	1542.67	678.07	230.94	631.80	194.67	58.07
3000	3000	3252	2749.02	2925.67	1589.72	672.16	236.35	626.34	281.51	57.55
3000	3000	0359	2838.11	3009.66	1536.34	660.67	199.31	616.25	180.18	56.64
3000	3000	5152	2813.72	2947.17	1460.81	667.53	234.72	622.11	232.56	57.18
3000	3000	9952	3039.31	3039.86	1481.78	702.42	212.16	654.60	187.76	60.19
10000	10000	1095	238140.14	447331.93	2182641.48	52639.81	20635.95	47413.95	20514.94	2374.29

Square, dense results on ***gruntу*** *from § 2.4.2.*

2.5 Conclusions

There are two limiting factors for multi-core standard simplex. Firstly, the standard simplex method itself is competitive only for problems which occur rarely in practice. Secondly, increasing the number of available cores does not necessarily increase the available memory bandwidth, and it is this bandwidth, rather than processing power, which limits the speed of such a solver. This makes it unlikely that there can be a standard simplex solver which is competitive for general problems on current hardware.

The many-core standard simplex implementation delivers surprising performance on the dense problem classes investigated. Contrary to previous reports of the inaccuracy of GPUs for mathematical programming [55], excellent results have been demonstrated in double precision arithmetic. When evaluating the performance of this code, however, it is important to note that the device in question is approximately twice as expensive as the workstation itself. The factor of ten performance on the largest problems must also be offset against the difficulty of constructing such a code.

Chapter 3

i7 - Parallel block-angular revised simplex

3.1 Overview

The revised simplex method has seldom been implemented in sparse arithmetic for parallel hardware. This may be attributed both to the scale of the undertaking, and also to the apparently indivisible and serial nature of the major subroutines of such a system.

The codes previously reviewed (see § 1.3.3) are predominantly task-parallel divisions of the algorithm, with each worker given a particular serial subroutine, or set of subroutines, to perform. Hall and McKinnon [60, 61] were able to identify as many as six such rôles, with the option to replicate some components for additional performance, but their approaches relied on deferred communication, and this caused stale data to be a source of inefficiency.

This chapter describes `i7`, a new, from scratch, data-parallel implementation of the revised simplex method, in sparse arithmetic, for commodity, multi-core hardware. It is based on two observations: firstly, that for problems with block structure, it is possible to decompose, into independent units, the majority of the computation involved in solves with the basis factorisation [74, 13]; secondly, that many real problems either have this form [69], or can be automatically brought into it using efficient partitioning techniques [38, 9].

`i7` is the first known implementation of the revised simplex method which maintains a compact, structured basis. Previous descriptions of such methods have either been entirely theoretical [83, 74, 13], confined to the standard simplex method [21], or have employed unstructured updates [86]. As such, this chapter is believed to contain both the first description of the application of modern simplex techniques to such a representation, and the first description of numerical and sparsity considerations in this setting.

This code is competitive in serial with established, non-commercial systems on some classes of problem, and is capable of reliably solving real-world problems. The structured form is shown to create only limited overhead, and speed-up of up to two on two processor packages (eight or sixteen cores) is demonstrated.

3.2 Kaul's method

The linear programming problem solved in practice is

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & Ax + s = 0 \\ & \ell \leq x \leq u \\ & -U \leq s \leq -L. \end{aligned} \tag{3.1}$$

Suppose the nonzeroes of the constraint matrix A are laid out as

$$\begin{bmatrix} A_{0,1} & \cdots & A_{0,r} \\ A_{1,1} & & \\ & \ddots & \\ & & A_{r,r} \end{bmatrix}, \tag{3.2}$$

a form which has been widely studied [9, 13, 26, 25, 31, 38, 74, 107, 115] and is called primal, or row-linked, block-angular form.

Two of the most expensive parts of an iteration of the revised simplex method are the solve with basis matrix B, termed FTRAN, and the solve with its transpose, BTRAN. Kaul [74], and independently Bennett [13], observed that for problems of the form (3.2), these operations may be largely decomposed by block.

Proposition 3.2.1 (Lasdon [83])**.** *B can always be permuted into the form*

$$\begin{bmatrix} B_{0,0} & B_{0,1} & \cdots & B_{0,r} \\ & B_{1,1} & & \\ & & \ddots & \\ & & & B_{r,r} \end{bmatrix}, \tag{3.3}$$

where $B_{i,i}$ for $i > 0$ have full row rank. All submatrices $B_{i,j}$ for $i, j > 0$ are drawn from the corresponding $[\, A_{i,j}\, I\,]$. The matrix $B_{0,0}$ is a selection of columns from the appropriate identity matrix.

Proof. B is made up of columns from $[\, A\, I\,]$, and the latter can be trivially permuted into the form

$$\begin{bmatrix} I_0 & & A_{0,1} & \cdots & & A_{0,r} \\ & I_1 & A_{1,1} & & & \\ & & & \ddots & & \\ & & & & I_r & A_{r,r} \end{bmatrix}. \tag{3.4}$$

The matrix B has full row rank by assumption of invertibility, and therefore all rows of B are linearly independent. Since each matrix $B_{i,i}$ for $i > 0$ is simply a set of rows of B, with only zero elements excluded, their rows are also linearly independent, and therefore they also have full row rank. □

As the blocks on the diagonal have full row rank, each contains a square, invertible submatrix of full height. Permuting any remaining columns to the

border, the working form of the structured basis,

$$B = \left[\begin{array}{cccc|ccc} S_0 & S_1 & \dots & S_r & C_1 & \dots & C_r \\ \hline & R_1 & & & T_1 & & \\ & & \ddots & & & \ddots & \\ & & & R_r & & & T_r \end{array}\right], \tag{3.5}$$

is obtained, where S_0 is $B_{0,0}$, T_i are square invertible matrices within $B_{i,i}$, R is the block rectangular remnant, and S_i and C_i are matching partitions of $B_{0,i}$.

3.2.1 Solving with B

Consider the forward solve with the basis matrix FTRAN, $Bx = b$. The matrix equation in the structured case is

$$\begin{bmatrix} S & C \\ R & T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{3.6}$$

Using the invertibility of T, the lower part can be solved for x_2, giving

$$x_2 = T^{-1}(b_2 - Rx_1). \tag{3.7}$$

Substituting into the upper part yields

$$(S - CT^{-1}R)x_1 = b_1 - CT^{-1}b_2 \tag{3.8}$$

Proposition 3.2.2 (Schur [104])**.** *The matrix $W = (S - CT^{-1}R)$ is invertible.*

Proof. Consider the matrix equation

$$\begin{bmatrix} S & C \\ R & T \end{bmatrix} \begin{bmatrix} \bar{S} & \bar{C} \\ \bar{R} & \bar{T} \end{bmatrix} = \begin{bmatrix} I & \\ & I \end{bmatrix} \tag{3.9}$$

where such $\bar{S}$, $\bar{C}$, $\bar{R}$ and $\bar{T}$ exist by invertibility of B. This implies

$$R\bar{S} + T\bar{R} = 0 \tag{3.10}$$
$$S\bar{S} + C\bar{R} = I \tag{3.11}$$

Solving (3.10) for $\bar{R}$ using the invertibility of T gives

$$\bar{R} = -T^{-1}R\bar{S}. \tag{3.12}$$

Substituting for $\bar{R}$ in (3.11) gives

$$(S - CT^{-1}R)\bar{S} = W\bar{S} = I. \tag{3.13}$$

Thus $\bar{S}$ is an inverse for W, as required. □

Using invertible representations of T and W it is therefore possible to solve for x in (3.6) by the sequence of operations given in Algorithm 3.2.1.

Now consider the solve with the transpose of the basis matrix BTRAN, $B^\top \pi = p$. The matrix equation in the structured case is

$$\begin{bmatrix} S^\top & R^\top \\ C^\top & T^\top \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}. \tag{3.14}$$

FTRAN(W,C,R,T,b)

1 **Solve** $Tz = b_2$ **for** z
2 **Form** $w = b_1 - Cz$
3 **Solve** $Wx_1 = w$ **for** x_1
4 **Form** $v = Rx_1$
5 **Solve** $Ty = v$ **for** y
6 **Form** $x_2 = z - y$
7 **Return** x

Algorithm 3.2.1: Kaul FTRAN

BTRAN(W,C,R,T,p)

1 **Solve** $T^\top y = p_2$ **for** y
2 **Form** $w = p_1 - R^\top y$
3 **Solve** $W^\top \pi_1 = w$ **for** π_1
4 **Form** $t = p_2 - C^\top \pi_1$
5 **Solve** $T^\top \pi_2 = t$ **for** π_2
6 **Return** π

Algorithm 3.2.2: Kaul BTRAN

Solving the lower part for π_2 gives

$$\pi_2 = T^{-\top}(p_2 - C^\top \pi_1), \tag{3.15}$$

and now substituting into the upper part yields

$$(S - CT^{-1}R)^\top \pi_1 = W^\top \pi_1 = p_1 - (T^{-1}R)^\top p_2, \tag{3.16}$$

so that invertible representations of T and W may be used to solve (3.14) by the sequence of operations given in Algorithm 3.2.2.

3.2.2 Factorisation

The conventional revised simplex method relies upon the availability of an invertible representation of the basis matrix B, with an LU factorisation usually preferred [89, 117, 112]. In contrast, Kaul's method, as it is described here, requires only the availability of invertible representations of the matrices T_i, and of the Schur complement W, but the effect of this change on the numerical stability and sparsity of the method has not been previously discussed.

Algorithms for LU factorisation are principally distinguished by the *pivoting scheme* they employ. *Partial pivoting* was shown to be numerically stable by Wilkinson [120], and the column-wise variant requires the largest element in the left-most column of the active submatrix be taken as the next pivot. As such, only the rows of the matrix need be re-ordered to match its factors.

When factorising simplex bases, however, performance considerations make maintaining the sparsity of the factors a major concern. An efficient implementation will perform an initial triangularisation phase [117], or use Markowitz pivoting [89, 112] to reduce fill-in, but these methods require the ability to re-order freely both the rows and columns of the active submatrix.

To see the effect of obtaining only a decomposed basis representation, consider the transposed and permuted matrix

$$PB^\top Q = \begin{bmatrix} T_1^\top & & & C_1^\top \\ & \ddots & & \vdots \\ & & T_r^\top & C_r^\top \\ R_1^\top & & & S_1^\top \\ & \ddots & & \vdots \\ & & R_r^\top & S_r^\top \\ & & & S_0^\top \end{bmatrix}. \tag{3.17}$$

The LU factors for each $T_i^\top$ form the principal diagonal of the factors for $PB^\top Q$, and the numerical stability of the basis representation as a whole clearly requires that the R_i be considered whilst factorising T_i. As the rows $R_i^\top$ share a common structure with the rows $T_i^\top$, exchanges may be made freely between the two matrices, which amounts to column exchanges in the original system. Thus for the left-hand side of $PB^\top Q$, column-wise partial pivoting is equivalent to row-wise partial pivoting in the rectangular systems $[\,T_i\,R_i\,]$.

Once the left-hand side of $PB^\top Q$ has been factorised, the remaining active submatrix is the transpose of the Schur complement W, but W is factorised directly in the decomposed representation, and so its transposed factors are interchangeable with those for the remaining submatrix of $PB^\top Q$. Thus the decomposed factors, when treated appropriately, result from column-wise partial pivoting on $PB^\top Q$, and working with the structured form of the basis does not have severe numerical consequences.

However, such a factorisation may contain additional nonzeroes. Pivots in columns on the right-hand side of (3.17) cannot be brought forward during triangularisation in the decomposed representation, regardless of merit count or the length of their column, and this can cause additional eliminations to be required.

3.2.3 Updates

To maximise the available parallelism in solves with the basis matrix, its structure must be maintained during updates. Consider the replacement of column p of B with an arbitrary column a of $[\,A\,I\,]$, giving $\bar{B}$. When B^{-1} is applied to this new matrix, the result is

$$B^{-1}\bar{B} = \begin{bmatrix} 1 & & \hat{a}_1 & & \\ & \ddots & \vdots & & \\ & & \hat{a}_p & & \\ & & \vdots & \ddots & \\ & & \hat{a}_m & & 1 \end{bmatrix}. \tag{3.18}$$

A structured update for B must restore the basis to the form (3.5), so that these additional nonzeroes $\hat{a}$ can be eliminated using only its decomposed representation. That the form (3.5) can always be induced was shown in Proposition 3.2.1, however the relationship between the structured forms which are available on two different iterations is less clear. Previous descriptions of this process have been based on mechanisms for problems with generalised upper bounds, have not maintained structure in R, and are numerically unstable [13, 83, 20].

The cases which occur are most easily distinguished by the location of the leaving column.

Loss in R_i The simplest case is that in which the leaving column is in block i of R in the current factorisation, so that T_i remains invertible. If the entering column is in another block j, R_i shrinks by one column, and R_j grows by one column. If the entering column is in the linking logicals S_0, these grow by one column and R_i again shrinks by one column. In any case, it is sufficient to note

the change in the columns of R and S and to update the factorisation of the Schur complement W.

Loss in S_0 The case when the leaving column is in the linking logicals S_0 is similar to the case in which it is in a block of R. If the entering column is in R_j then R_j grows by one column, and S_0 shrinks by one column. It is again sufficient to note the change in R and S and to update the factorisation of the Schur complement W.

Loss in T_i When the leaving column is in T_i then maintaining the structure of the basis matrix is significantly more difficult. Each T_i is a self-contained, invertible matrix and the tableau column $\hat{a}$ is calculated with the inverse of another matrix B, so that even when the entering column is in the same block, $\hat{a}$ does not enter T_i directly. In that case, the pivotal column in T is instead $T_i^{-1}\ell$, where ℓ is the lower part of the incoming column, found as a byproduct of the structured FTRAN.

Although the matrix B remains nonsingular throughout the solve, which ensures $\hat{a}_p \neq 0$, the only guarantee for T_i is that some collection of the columns in $[\,T_i\,R_i\,]$ always form a nonsingular matrix. As such, it is possible to lose a column from T_i without gaining a replacement in the same block, and even when there is an entering column in the same block, it is not always the case that, for ψ the index of the leaving column in T_i, the element $e_\psi^\top T_i^{-1}\ell \neq 0$.

These considerations mean that a column leaving T_i requires the pivots $e_\psi^\top T^{-1}r$ to be evaluated for every column r in R_i. When the entering column is in the same block, $e_\psi^\top T^{-1}\ell$ must also be considered. Although structurally at least one of these elements must be nonzero, for numerical stability the column selected to enter T_i must at least pass some threshold test on its candidate pivot.

In every case, the factorisations of T_i and W are updated. If the entering and leaving columns are in different blocks, R_i shrinks and R_j grows, and the change in R and S is noted. If the entering column is not selected to enter T_i, the basis matrix must also be permuted to bring the required element of R_i into T_i.

3.3 The block-diagonal, invertible matrix T

The matrix T in (3.5) can be decomposed into a set of submatrices T_i of fixed dimension, and the independence of these matrices allows operations with T to be parallelised. For problems in which T comprises most of the basis B, it seems possible that this parallelism will compensate for the additional complexity and basis fill incurred by the structured form.

Each matrix T_i behaves very much as a smaller version of B in a typical implementation of the revised simplex method. It is amenable to all of the same optimizations, and requires the same operations to be provided. The only differences are in the factorisation step, which is considered first.

3.3.1 Factorisation

The numerical considerations which affect the factorisation of T were described in § 3.2.2, where it was noted that stability requires pivots from both T_i and

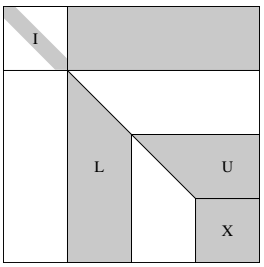


Figure 3.1: Triangularisation of the basis; shaded areas are structurally nonzero.

R_i to be available. However, a practical routine must also seek to promote sparsity, and the following is believed to be the first description of the special considerations which apply to T in this context.

Consider factorising an invertible matrix B. For any candidate pivot b_{ij}, if there are r_i nonzeroes in row i and c_j nonzeroes in column j of the active submatrix, then its Markowitz merit score μ_{ij} [89] is defined as

$$\mu_{ij} = (r_i - 1)(c_j - 1). \tag{3.19}$$

Proposition 3.3.1 (Markowitz [89])**.** *The score μ_{ij} is an upper bound on the fill-in created by the choice of b_{ij} for the next pivot.*

Proof. When b_{ij} is selected for pivoting, all other nonzeroes in row i of the active submatrix must be eliminated by subtracting a multiple of column j at these positions. This creates a maximum of $(r_i - 1)(c_j - 1)$ new nonzeroes. □

Clearly, for a nonzero which is alone in its column or row, the Markowitz bound is zero, which makes it sufficient to permute these entries to the diagonal without performing any eliminations. This process is called *triangularisation*, and is an essential part of an efficient routine for the LU factorisation of simplex bases [87]. The resulting matrix form is shown in Figure 3.3.1 [117], and the unstructured remainder X, the *bump*, is usually small.

When triangularisation is applied to a matrix T_i, the effect of pivots on the other constituent parts of B must also be considered. A column which has a single entry in the active submatrix of $[\,T_i\,R_i\,]$ may have additional, unavailable entries in C_i or S_i. Although disregarding such nonzeroes will not create fill in the factors of T_i, it will lead to increased fill in W. Row singletons must similarly be evaluated across the entirety of $[\,T_i\,R_i\,]$, not only because of fill in W, but also because it is unknown which particular columns will comprise T_i.

3.3.2 Updates

The update of the factorisation of each T_i is much the same as the update of the factorisation of a general basis matrix B. Despite the surrounding complexity, only two situations can obtain on a given iteration for a given T_i: a column may be replaced, or the matrix may be unchanged. As such, the techniques available for updating the factorisation are well known, and are only briefly reviewed here.

The revised simplex method was originally described with a dense, explicit inverse [32], but this requires $O(m^2)$ operations to update and is frequently full.

Product form updates [30] can be applied to any representation of an inverse in $O(m)$ time, but are prone to numerical instability and fill. Most modern codes update the LU factors directly, in a manner first described by Bartels and Golub [11, see also 10], and subsequently improved on by Forrest and Tomlin [42] and later Suhl and Suhl [111]. The performance of such updates is analysed in detail in Appendix D.

3.3.3 Efficient operations

For a dense right-hand side, the solves $Tz = b$ and $T^\top \pi = t$ decompose into independent operations with each T_i. When the right-hand side is structured, however, only one T_i operates on nonzeroes, which has particular significance in practice for Algorithm 3.2.1 (FTRAN). This makes it essential to employ efficient techniques for solves with a sparse right-hand side.

Proposition 3.3.2 (Gilbert and Peierls [45])**.** *Let L be a lower triangular matrix and $G = (V, E)$ be a directed graph, where $V = 1 \dots n$ and*

$$E = \{\, (i, j) \;\; s.t. \;\; \ell_{ji} \neq 0 \,\}. \tag{3.20}$$

Let K be the set of nonzero indices of b, and Z be the set of nonzero indices of x. Then the set of vertices reachable by a path through G from K contains Z.

Proof. For the first index,

$$x_1 = \frac{b_1}{\ell_{11}} \tag{3.21}$$

which is nonzero iff $b_1 \neq 0$, so that either $1 \in K$ and $1 \in Z$, or $1 \notin K$ and $1 \notin Z$. Clearly the proposition holds for the first index.

Consider any index j such that the proposition holds for all $i < j$. Now

$$x_j = \frac{b_j}{\ell_{jj}} - \sum_{i<j} x_i \frac{\ell_{ji}}{\ell_{jj}}, \tag{3.22}$$

so that, as for the first index, if $b_j \neq 0$ then $j \in K$ so the proposition holds.

Suppose $b_j = 0$ and $x_j \neq 0$, then at least one term $x_i \ell_{ji}$ is nonzero, say $x_p \ell_{jp}$. By induction, there is a path from K through G to p, and furthermore $\ell_{jp} \neq 0$ so that (p, j) is an edge in G. Combining the path to p through G from K with the edge (p, j) gives a path from K through G to j. □

The graph G is called the *elimination graph* on L. Notice that in Proposition 3.3.2, the set of potential nonzeroes reachable from K through G contains only elements of x which are nonzero when cancellation is excluded. The size of this set is a bound on the number of nonzeroes in x which is typically very close to being tight in practice.

The method of Gilbert and Peierls [45] can be used to identify the elementary matrices which will need to be applied, ahead of performing FTRAN with triangular factors on a sparse right-hand side. By traversing the paths from K, it generates the nonzero indices of the result vector in *topological order*, which ensures that for all indices j reachable through G from an index i, j occurs subsequently to i.

Such a list of indices can then be translated into a list of matrices in the order they should be applied. The analysis step creates significant additional overhead due to the topological search which is performed, but can be abandoned when too many nonzeroes have been identified in the result.

3.4 The Schur complement W

The Schur complement W plays an essential rôle in solves with the decomposed representation of B, but its dependence on both the inverse of T and also the columns on the left-hand side of (3.5) makes the corrections to its invertible representation between iterations more complex than those for a traditional basis matrix. This section describes a product form update, which is believed to be novel, that enables a factored form to be maintained efficiently.

Throughout this section, it is assumed that an entering column may replace any column on the left-hand side of (3.5) directly. The block-rectangular structure of R is maintained separately by a permutation matrix Q, which will be discussed in a later part.

3.4.1 Updates

Consider an update to B taking the form

$$\bar{B} = B + (a_q - b_p)e_p^\top, \tag{3.23}$$

where the leaving column b_p falls on the right-hand side of B, in C and T. If the upper and lower parts of the constraint matrix A are denoted by U and L respectively, matching the division of B into an upper part of C and S and a lower part of T and R, then this update can also be written as

$$\bar{C} = C + (u_q - c_\psi)e_\psi^\top \tag{3.24}$$

$$\bar{T} = T + (\ell_q - t_\psi)e_\psi^\top, \tag{3.25}$$

where ψ is the index of the leaving column in C and T.

The matrices S and R are unchanged, so that the updated W is given by

$$\bar{W} = S - \bar{C}\bar{T}^{-1}R. \tag{3.26}$$

The term $\bar{T}^{-1}$ can be expressed in terms of T^{-1}, using the Sherman-Morrison formula [105], giving

$$\begin{aligned} \bar{T}^{-1} &= \left(T + (\ell_q - t_\psi)\, e_\psi^\top\right)^{-1} \\ &= \left(I - \frac{\tau - e_\psi}{e_\psi^\top \tau} e_\psi^\top\right) T^{-1}. \end{aligned} \tag{3.27}$$

where $\tau = T^{-1}\ell_q$ is the incoming column in T, as used in the update of T itself.

Substituting into (3.26) now yields

$$\begin{aligned} \bar{W} &= S - \left(C + (u_q - c_\psi)e_\psi^\top\right)\left(I - \frac{\tau - e_\psi}{e_\psi^\top \tau} e_\psi^\top\right) T^{-1}R \\ &= W + w_1 v^\top, \end{aligned} \tag{3.28}$$

with

$$w_1 = (u_q - c_\psi) + \bar{C}\left(\frac{\tau - e_\psi}{e_\psi^\top \tau}\right) \tag{3.29}$$

$$v^\top = e_\psi^\top T^{-1}R. \tag{3.30}$$

Suppose now that the the leaving column, say z, falls on the left-hand side of B, in S and R. The update becomes

$$\bar{S} = S + (u_q - s_\zeta)e_\zeta^\top \tag{3.31}$$

$$\bar{R} = R + (\ell_q - r_\zeta)e_\zeta^\top, \tag{3.32}$$

where ζ is the index of z in R and S, so that

$$\begin{aligned} \bar{W} &= \bar{S} - CT^{-1}\bar{R} \\ &= W + w_2 e_\zeta^\top, \end{aligned} \tag{3.33}$$

with

$$w_2 = (u_q - s_\zeta) - C\left(\tau - T^{-1}r_\zeta\right). \tag{3.34}$$

The most complex case is that in which a permutation is required to bring a column into T from R, in order to maintain the structure of the basis. This has the form

$$\bar{B} = B + (a_q - b_z)e_z^\top + (b_z - b_p)e_p^\top, \tag{3.35}$$

with p on the right-hand side of B and z on the left-hand side, so that

$$\bar{S} = S + (u_q - s_\zeta)e_\zeta^\top \tag{3.36}$$

$$\bar{C} = C + (s_\zeta - c_\psi)e_\psi^\top \tag{3.37}$$

$$\bar{R} = R + (\ell_q - r_\zeta)e_\zeta^\top \tag{3.38}$$

$$\bar{T} = T + (r_\zeta - t_\psi)e_\psi^\top. \tag{3.39}$$

The update for W follows mechanically from

$$\begin{aligned} \bar{W} = &\left(S + (u_q - s_\zeta)e_\zeta^\top\right) \quad - \\ &\left(C + (s_\zeta - c_\psi)e_\psi^\top\right)\left(I - \left(\frac{\tau - e_\psi}{e_\psi^\top \tau}\right)e_\psi^\top\right)T^{-1}\left(R + (\ell_q - r_\zeta)e_\zeta^\top\right) \\ &= W + w_1 v^\top + (w_2 + w_3)e_\zeta^\top, \end{aligned} \tag{3.40}$$

where w_2 and v are as before, s_ζ and r_ζ are used in place of u_q and ℓ_q in the definition of w_1, and

$$w_3 = e_\psi^\top\left(\tau - T^{-1}r_\zeta\right)w_1. \tag{3.41}$$

Equation (3.40) contains (3.28) and (3.33) as special cases, and will henceforth be taken as the standard form.

An update is sought for an invertible representation of W, not the matrix itself. Let $\omega_i = W^{-1}w_i$ then, using the Woodbury formula [123], the updated $\bar{W}^{-1}$ can be written

$$\begin{aligned} \bar{W}^{-1} = W^{-1} - &\begin{bmatrix} \omega_1 & (\omega_2 + \omega_3) \end{bmatrix} \cdot \\ &\left(I + \begin{bmatrix} v & e_\zeta \end{bmatrix}^\top \begin{bmatrix} \omega_1 & (\omega_2 + \omega_3) \end{bmatrix}\right)^{-1} \begin{bmatrix} v & e_\zeta \end{bmatrix}^\top W^{-1} \end{aligned} \tag{3.42}$$

The embedded 2×2 matrix inverse, say K^{-1}, can be found by the standard formula, giving

$$\begin{aligned} K^{-1} &= \begin{bmatrix} 1 + v^\top\omega_1 & v^\top(\omega_2 + \omega_3) \\ e_\zeta^\top\omega_1 & 1 + e_\zeta^\top(\omega_2 + \omega_3) \end{bmatrix}^{-1} \\ &= \frac{1}{\lambda}\begin{bmatrix} 1 + e_\zeta^\top(\omega_2 + \omega_3) & -v^\top(\omega_2 + \omega_3) \\ -e_\zeta^\top\omega_1 & 1 + v^\top\omega_1 \end{bmatrix}, \end{aligned} \tag{3.43}$$

where

$$\lambda = \left(1 + e_\zeta^\top(\omega_2 + \omega_3)\right)(1 + v^\top\omega_1) - v^\top(\omega_2 + \omega_3)e_\zeta^\top\omega_1. \tag{3.44}$$

It follows that there exists a product form update for the inverse of the Schur complement, of the form

$$\bar{W}^{-1} = \left(I - \frac{1}{\lambda}fv^\top - \frac{1}{\lambda}ge_\zeta^\top\right)W^{-1}, \tag{3.45}$$

where

$$f = \omega_1\left(1 + e_\zeta^\top(\omega_2 + \omega_3)\right) - (\omega_2 + \omega_3)e_\zeta^\top\omega_1 \tag{3.46}$$

$$g = (\omega_2 + \omega_3)(1 + v^\top\omega_1) - \omega_1 v^\top(\omega_2 + \omega_3). \tag{3.47}$$

Note that the vector ω_2 can be written

$$\begin{aligned}\omega_2 &= W^{-1}\Big((u_q - s_\zeta) - C\left(\tau - T^{-1}r_\zeta\right)\Big)\\ &= W^{-1}\left(u_q - C\tau\right) - e_\zeta,\end{aligned} \tag{3.48}$$

and is hence a byproduct of the structured FTRAN, in which $Wx = (u_q - C\tau)$ is solved. Similarly, the vector v is available as the active row of $T^{-1}R$, which is required for pivoting in the structured form (see § 3.2.3). The vector ω_3 is a multiple of ω_1 found from the change in the pivot in row ψ of T. The vector ω_1, however, is not easily available, and requires an additional FTRAN with W to be performed on each iteration in which T changes.

3.4.2 Solves with W

For FTRAN,

$$Wx = b, \tag{3.49}$$

then if W^{-1} has a product form representation

$$W^{-1} = \prod_{i=1}^{k}\left(I - \frac{1}{\lambda_i}f_i v_i^\top - \frac{1}{\lambda_i}g_i e_{\zeta_i}^\top\right) \tag{3.50}$$

the solve can be performed with each term in order as

$$x = b - \frac{(v^\top b)}{\lambda}f - \frac{(e_\zeta^\top b)}{\lambda}g. \tag{3.51}$$

For BTRAN,

$$W^\top x = b, \tag{3.52}$$

the same product form representation is transposed to give

$$W^{-\top} = \prod_{i=k}^{1}\left(I - \frac{1}{\lambda_i}v_i f_i^\top - \frac{1}{\lambda_i}e_{\zeta_i}g_i^\top\right), \tag{3.53}$$

and the solve can be performed with each term in reverse order as

$$x = b - \frac{(f^\top b)}{\lambda}v - \frac{(g^\top b)}{\lambda}e_\zeta. \tag{3.54}$$

3.5 The permutation matrix Q

The matrix R in (3.5) is composed of columns of two types, zero columns below the linking logicals S_0, and overflow columns from T which will share the structure of some T_i. This means that R can be permuted into a block-rectangular form, although the number of columns in each R_i will vary between basis matrices, and some blocks may be empty. Previous descriptions of Kaul's algorithm have not included the steps required to maintain this structure in R, but doing so provides the computational advantage of localising to well-defined, disjoint ranges, the values corresponding to variables from the different blocks.

The Schur complement W shares a column ordering with R, and cannot be permuted directly, owing to the complexity of its factors. Instead, a permutation matrix Q is added to the representation, so that $W = W'Q$. After each refactorisation, $Q = I$ and $W = W'$, but in the column ordering of W', an entering column in S and R takes the place of its corresponding leaving column, with Q being updated to bring $W'Q$ back to the same ordering as R. This section provides a description of techniques, which could not all be identified in the literature, for performing efficient solves on sparse vectors with a permutation matrix Q.

3.5.1 Updates

Permutation matrices can be stored efficiently as vectors of integers

$$Q = [\, q_1 \dots q_n \,]\,. \tag{3.55}$$

This representation has two mutually inverse interpretations: as "goes to", where in $y = Qx$, the element $y_{q_i} = x_i$, so that q_i is the index of the column containing a nonzero in row i; or as "comes from", so that $y_i = x_{q_i}$, and q_i is the index of the row containing a nonzero in column i.

If Q is pre-multiplied by another permutation matrix, hence permuting the rows of Q, then in the first form ("goes to") the indices in the vector representation of Q can be permuted directly, which is inexpensive. In contrast, in the second form ("comes from") a search must be conducted for the row in which each column has its nonzero. When Q is post-multiplied by another permutation matrix, hence permuting its columns, then the relative costs of the two forms are reversed.

Columns will be assumed enter or leave a block of R at its right-hand edge, so that a leaving column will first be flipped to the last position in its block. If one column leaves R_i and another enters R_j, then there will be a shift in the blocks between R_i and R_j. If $i < j$, the columns of R_k, for $i < k \leq j$, will shift left, while if $i > j$, the columns of R_k, for $j < k \leq i$, will shift right, and the cases involving S_0 are obvious extensions. There are thus two types of update to Q, post-multiplication by a transposition which moves a leaving column to the right-hand margin of its block, and post-multiplication by a simple, but potentially very long, cycle which shifts a set of blocks left or right.

If Q is post-multiplied by a permutation, then Q^{-1} is pre-multiplied by its inverse. By the previous logic, Q can be maintained efficiently in the second form, and Q^{-1} can be maintained efficiently in the first form. Since the two forms are mutually inverse, there is only one viable permutation matrix which can be updated in practice.

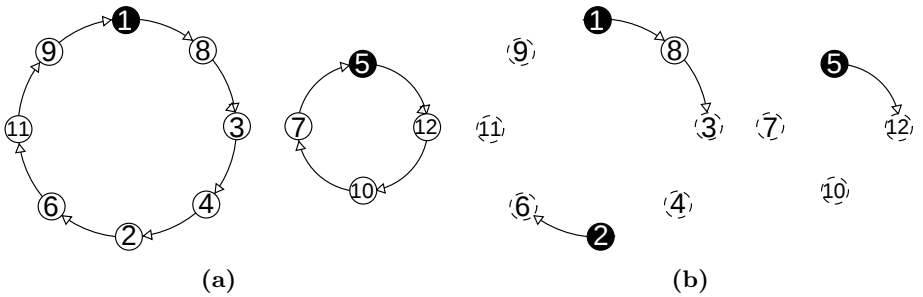


Figure 3.2: (a) The permutation [8 6 4 2 12 11 5 3 1 7 9 10] as a pair of cycles. Black nodes represent the entry points for the algorithms given. (b) The same permutation, but with an example of truncated pursuit where the dashed nodes hold zeroes.

3.5.2 Solves with Q

The simplest method of solving with Q is to select elements into a scratch vector, and then copy the result back over the input. This is inefficient even when the input vector is dense, as the cost of $2n$ floating point writes and $2n$ floating point reads is independent of the complexity of the permutation.

If the input vector y is sparse, so that a set of nonzero positions Z_y is given, then the solve $Q^\top x = y$ (BTRAN) can be done more efficiently as

$$x_{q_i} = \begin{cases} y_i & i \in Z_y \\ 0 & \text{otherwise.} \end{cases} \tag{3.56}$$

However, the solve $Qx = y$ (FTRAN) admits no efficient implementation, as in

$$x_i = \begin{cases} y_{q_i} & q_i \in Z_y \\ 0 & \text{otherwise} \end{cases} \tag{3.57}$$

each index i must be permuted to determine whether the corresponding entry of y is nonzero.

An alternative is approach *cycle pursuit*. Recall that every permutation is expressible as a product of disjoint cycles, where a cycle is an ordered subset of indices through which values are rotated upon each application of the permutation, shown graphically in Figure 3.2.

By following each cycle in turn, it is possible to permute a vector of any length using only floating point storage of constant size, though an array of n booleans is still needed to mark completed elements. Methods avoiding the use of temporary storage modify the data itself, for example by sign flipping, but this is impractical for floating point vectors [77].

The identity permutation is an empty product of cycles, and it may be hoped that the complexity of a permutation as a product of cycles is related to the true complexity of the permutation itself. The details for the dense case are given in Algorithm 3.5.1 and Algorithm 3.5.2. The algorithm for BTRAN is well known (e.g. [39, 77]). The algorithm for FTRAN could not be found in this form in the literature.

Compared to the copy approach, the number of floating point reads and writes is now n each, if all scalar variables are assumed to be held in registers,

Q-FTRAN(Q, x, m, b)

```
1  ▷ Q is the permutation
2  ▷ x is the input
3  ▷ m is the base dimension
4  ▷ b is scratch space
5  b ← FALSE
6  for r ∈ 1...m do
7        if b_r = TRUE then
8              continue
9        endif
10       s ← x_r
11       c ← r
12       n ← q_c
13       while n ≠ r do
14             x_c ← x_n
15             b_n ← TRUE
16             c ← n
17             n ← q_c
18       done
19       x_c ← s
20       b_r ← TRUE
21 done
```

Algorithm 3.5.1: Dense FTRAN

Q-BTRAN(Q, x, m, b)

```
1  ▷ Q is the permutation
2  ▷ x is the input
3  ▷ m is the base dimension
4  ▷ b is scratch space
5  b ← FALSE
6  for r ∈ 1...m do
7        if b_r = TRUE then
8              continue
9        endif
10       s ← x_r
11       c ← r
12       n ← q_c
13       while n ≠ r do
14             SWAP(x_n, s)
15             b_c ← TRUE
16             c ← n
17             n ← q_c
18       done
19       x_r ← s
20       b_c ← TRUE
21 done
```

Algorithm 3.5.2: Dense BTRAN

and the number of boolean operations is at most $2n$ reads and n writes. In the best case, the identity permutation, there are n floating point reads, n boolean reads, and n boolean writes.

When the input vector is sparse, the cycle pursuit can be truncated to give a more efficient solve. For this, it is necessary to have available an array of n integers and floating point storage of constant size. In the case of Algorithm 3.5.4 (BTRAN), the work done is always $O(z)$ and essentially optimal because cycle-following can be terminated at the first zero. This is not possible in Algorithm 3.5.3 (FTRAN) because the final position of the first nonzero is unknown before the cycle is complete. Descriptions of these algorithms could not be identified in previously published material.

3.6 Structured programs

For its basis matrices to take on the structured form (3.5), a program must itself be in block-angular form, and the location of those blocks known. If a code creates a linear programming formulation from a higher-level representation, then such information is easily available, but even if a constraint matrix is all that is given, it is still usually possible to identify some structure in a problem.

Multi-commodity flow problems arise when several different quantities traverse a shared network with capacity constraints on its arcs, and their formulations as linear programs are primal block-angular, with one block for each commodity [25]. The duals of stochastic programming problems [86] constitute

Q-FTRAN(Q, x, z, ℓ, m, h)

```
1  ▷ Q is the permutation
2  ▷ x is the input
3  ▷ z are the nonzero indices
4  ▷ ℓ is the number of nonzeroes
5  ▷ m is the base dimension
6  ▷ h is scratch space
7  for i ∈ 1...ℓ do
8        h_{z_i} ← i
9  done
10 for r ∈ 1...ℓ do
11       if h_{z_r} < 0 then
12             continue
13       endif
14       s ← x_{z_r}
15       c ← z_r
16       n ← q_c
17       while n ≠ z_r do
18             x_c ← x_n
19             if x_c ≠ 0 then
20                   z_{h_n} ← c
21                   h_c ← −h_n
22             endif
23             c ← n
24             n ← q_c
25       done
26       x_c ← s
27       z_r ← c
28       h_c ← −r
29 done
```

Algorithm 3.5.3: Sparse FTRAN

Q-BTRAN(Q, x, z, ℓ, m, h)

```
1  ▷ Q is the permutation
2  ▷ x is the input
3  ▷ z are the nonzero indices
4  ▷ ℓ is the number of nonzeroes
5  ▷ m is the base dimension
6  ▷ h is scratch space
7  for i ∈ 1...ℓ do
8        h_{z_i} ← i
9  done
10 for r ∈ 1...ℓ do
11       if h_{z_r} < 0 then
12             continue
13       endif
14       s ← x_{z_r}
15       t ← h_{z_r}
16       x_{z_r} ← 0
17       c ← z_r
18       n ← q_c
19       while n ≠ z_r and s ≠ 0 do
20             SWAP(x_n, s)
21             SWAP(h_n, t)
22             if x_n ≠ 0 then
23                   z_t ← n
24                   h_n ← −h_n
25             endif
26             c ← n
27             n ← q_c
28       done
29       if s ≠ 0 then
30             x_{z_r} ← s
31             h_{z_r} ← −t
32             z_t ← z_r
33       endif
34 done
```

Algorithm 3.5.4: Sparse BTRAN

another standard class of such formulations. More generally, when a modelling language representation of a problem is available, it may be possible to infer block structure from the index sets which are used [115].

Several methods of identifying, or imposing, primal block-angular structure on a bare constraint matrix exist. The bipartite graph partitioning approach [38] maps each row and column to a vertex, and each nonzero to an edge connecting such vertices. When the vertices are split into blocks so as to minimise the number of cut edges, the result can be used to structure A. Hypergraph partitioning [9] can be used similarly to bring A into the desired form.

3.6.1 Multi-commodity flow problems

A multicommodity flow problem consists of a network across which products of different types must be transported, at given rates, as cheaply as possible. It can be described in JLF format [69], which provides minor generalisations of the standard form, as discussed below. Parts of the analysis which follows are believed to be novel, as no parser capable of automatically inferring the full problem specification from JLF files is known to have been previously described or implemented.

In JLF format, flows through the network may be identified by the type of product they represent, by the node from which the particular products originated, by the node to which the particular products are bound, or by some combination of these properties. These specifications may vary in different parts of the network, so that a flow which was distinguished only by its product type, in one region, may instead need to be labelled by its ultimate origin upon entering another.

The connectivity of the network is given only in terms of *links*, each of which is the maximum rate at which some flow specified in the previous way may pass between pairs of nodes, and the cost per unit of such transport. Similarly, this type of flow specification is used to give the rate of supply and demand at nodes in the network. Shared capacity constraints are provided by *bundles*, which may also be used to provide bounds on the total quantity of a set of products that can be simultaneously transported across a group of arcs.

A distinguishable type of flow in this representation is called a *commodity*, and is a triple formed from an origin, destination and product type. The constraints for the linear programming formulation of a JLF instance contain a network incidence matrix for each commodity, forming a block-angular part, and linking rows for the shared capacity specifications. A parser for JLF can thus be used in turn to generate naturally primal block-angular problems for solution.

This format was intended to accommodate several disparate input formats used by existing multi-commodity solvers, by providing a superset of their allowed representations. Each solver could assume that its particular dialect of JLF had been used for any input it received, and any tool working with general JLF files could be told their orientation (see [69] for more details). The standard dialects of JLF formulate commodities as origins, destinations, origin-destination pairs, or products. The implementation of a parser for arbitrary files in JLF format is thus non-trivial.

The JLF specification mandates that a link is repeated to produce an arc for every distinguishable commodity to which it might apply, and as each such arc may have its own capacity constraint, the interpretation of the problem is fundamentally altered by the number of identifiable commodities. It is also possible for JLF files to specify that a particular number of products exist, independent of any other considerations. This necessitates an analysis step, and finding the fewest number of identifiable commodities leads to definitions consistent with the standard JLF dialects.

Significant complexity arises in a general parser because of the power of the JLF representation. It is possible, for example, to frame problems with both multi-layer, and mixed-mode commodities, so that the standard dialects incorporate only a small fraction of the allowable input.

3.6.2 Bipartite graph partitioning

Consider the undirected graph $G = (V, E)$ derived from a matrix $A \in \mathbb{R}^{m \times n}$ by

$$\begin{aligned} V &= \{\, (i, \mathbf{r}) \mid i \in [\,1 \ldots m\,]\,\} \cup \{\, (i, \mathbf{c}) \mid i \in [\,1 \ldots n\,]\,\} \\ E &= \big\{\, \{\, (i, \mathbf{r}), (y, \mathbf{c})\,\} \mid a_{ij} \neq 0 \big\} \end{aligned} \tag{3.58}$$

where $\mathbf{r}$ and $\mathbf{c}$ are tokens to distinguish vertices originating from rows of the constraint matrix and vertices originating from its columns.

Suppose that the vertices of G are divided into r disjoint sets. An edge is said to be cut by this partition if its endpoints fall into different sets.

Proposition 3.6.1 (Ferris and Horn [38])**.** *If no edges in E are cut by a partition P of G, then A can be permuted into block-diagonal form.*

Proof. Consider an off-diagonal nonzero a_{ij}, represented by the edge $\{(i, \mathbf{r}), (j, \mathbf{c})\}$. If $(i, \mathbf{r})$ and $(j, \mathbf{c})$ belong to different sets in the partition, then this edge is cut, contradicting our assumption. Thus for every nonzero, the vertices for its row and column belong to the same set. If the matrix is permuted to bring together rows and columns with vertices in the same set of the partition, and these sets occur in the same order top-to-bottom for the rows as they do left-to-right for the columns, then the matrix is block-diagonal, as required. □

Suppose that all cut edges are incident on vertices in the first set, say with index zero, of the partition. If the permutation of Proposition 3.6.1 is applied to bring rows and columns in the same set together, then the structure that results is called *arrowhead form*, and the first set is called the *border*.

The method of Ferris and Horn [38] begins by partitioning the bipartite graph of (3.58) into some pre-selected number of sets, say $1 \ldots r$, with an initially empty border. A greedy heuristic is then applied which repeatedly moves to the border the vertex upon which the most cut edges are incident. An edge is not considered cut if it is already incident on some vertex in the border. This continues until there are no more such edges.

If the set of column vertices in the border is empty, the previously described permutation will bring the matrix into primal block-angular form. If not, the matrix is expanded by dividing each column whose vertex is in the border, and creating a new row to link the two parts. For example, to transform the matrix

$$\begin{aligned} a_{1,3}x_3 + a_{1,1}^\top x_1 \qquad\qquad &= b_1 \\ a_{2,3}x_3 \qquad\qquad + a_{2,2}^\top x_2 &= b_2 \end{aligned} \tag{3.59}$$

into primal block-angular form

$$\begin{aligned} x_3 - \qquad\qquad x_4 \qquad\qquad &= 0 \\ a_{1,3}x_3 + a_{1,1}x_1 \qquad\qquad &= b_1 \\ a_{2,4}x_4 + a_{2,2}x_2 &= b_2 \end{aligned} \tag{3.60}$$

a new variable x_4 and a new constraint can be introduced.

3.6.3 Hypergraph partitioning

The nonzeroes of a matrix $A \in \mathbb{R}^{m \times n}$ have a natural representation as a hypergraph $H = (V, E)$ over its columns, where

$$\begin{aligned} V &= [\,1 \ldots n\,] \\ E &= \big\{\, \{\, j \mid a_{ij} \neq 0 \,\} \;\big|\; i \in [\,1 \ldots m\,] \,\big\}. \end{aligned} \tag{3.61}$$

For example the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & & & a_{16} \\ & a_{22} & & & & a_{26} \\ a_{31} & & a_{33} & a_{34} & & \\ a_{41} & & & a_{44} & & \\ & a_{52} & & & a_{55} & a_{56} \end{bmatrix}, \tag{3.62}$$

in which the omitted entries are zero, can be represented by the hypergraph shown in Figure 3.3(a).

When H is partitioned, the result is to place its vertices in disjoint sets, and hence to assign each column to a particular block. A hyperedge is said to be cut by a partition if it is incident on vertices from more than one set, meaning that its row has nonzeroes in more than one block, and partitioning can be carried out to minimise cut weight.

If all hyperedges cut by a partition are removed from consideration, then the remaining rows by definition have nonzeroes in columns from a single block. Aykanat et al. [9] assign uncut hyperedges to the same sets as the vertices they are incident upon, and hence their corresponding rows to the same block as the columns in which they have nonzeroes. Cut hyperedges, representing rows which have nonzeroes linking two blocks, are ignored, and the corresponding row is moved to the border. When columns and rows in the same block are permuted together, as for Proposition 3.6.1, the resulting form is primal block-angular. Figure 3.3(b) shows the effect on the example matrix of this procedure.

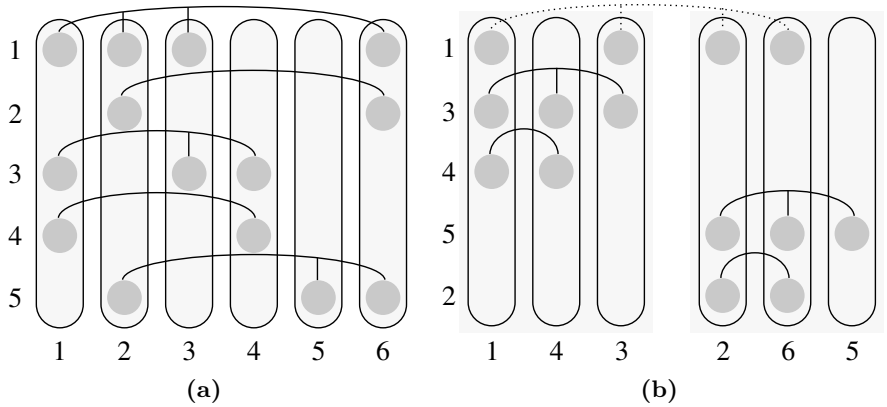


Figure 3.3: (a) A hypergraph derived from the nonzero pattern of a matrix. (b) The partitioned graph can be used to impose structure on the matrix.

3.7 Parallel revised simplex

i7 is an implementation of the primal, column-wise, revised simplex method in sparse arithmetic, making use of Kaul's method to provide parallel operations with the factorisation of the basis matrix. It is written for multi-core machines, and uses Posix thread primitives.

3.7.1 Algorithmic elements

A bound-breaking phase I ratio test Following [91], during row selection in phase I new infeasibilities may be introduced, provided that the sum of infeasibilities is decreased. This measure generally improves both the time taken to find a feasible point and the numerical stability of the solver.

A two pass phase II ratio test Following [63], two passes are taken in the ratio test, with the feasible region slightly expanded during the first. The maximum step in this enlarged region is computed, and used as a limit for the row selected in the subsequent pass.

Approximate steepest edge pricing Following [113], each column's norm is initially taken as the number of nonzero entries it contains in the constraint matrix. Subsequent updates simply ignore the expensive inner product upon which a true steepest edge calculation depends, on the grounds that for sparse vectors, it may be expected to be near zero.

Wolfe's ad-hoc method for anti-cycling Following [122], when a stall is detected, an explicit, symbolic perturbation is applied to all of the degenerate positions. This is a method of guaranteed correctness, which cannot cycle, in contrast to the more popular Expand method [46] which is faster.

Elimination form inverses with product form updates Following [89], the invertible representations of T and W are created initially by LU factorisation. After each iteration, standard rank 1 product form updates are applied to the representations of T_i^{-1}, and rank 2 product form updates are applied to the representation of W^{-1}.

Fast symbolic invert Following [117], all identity columns in the basis are ignored, then row and column singletons are extracted by a fast triangularisation process. The remainder is initially sorted either row-wise or column-wise by Markowitz merit count, then left-looking partial pivoting in the chosen direction is applied. For difficult matrices, a switch is made to dense, right-looking, full pivoting for the final pivots.

Sparse algebra in which non-zeroes are tracked This is a well known technique, e.g. [125], in which sparse vectors are stored full length, but a separate list of non-zeroes is maintained. As problems become less dense, the importance of knowing the non-zeroes increases: for example 1% density or lower is not uncommon for the working vectors encountered when solving some problems.

Hyper-sparse FTRAN by graph traversal Following [45], the etas in the LU factors which must be applied to a sparse right-hand side are pre-calculated, when appropriate, by a depth first search. This is a significant saving in many cases, as the vast majority of the eta matrices are both from the inverse and also unnecessary to apply.

Sparse column selection Following [62], a list of attractive incoming columns is maintained, so that unattractive positions need not be traversed during column selection.

Sparse tableau row generation Following [62], the constraint matrix is maintained row-wise, permuted into the form $[\,N\,B\,]$. When the nonzeroes in the result of BTRAN, π, are tracked, the number of floating point operations needed to find the pivotal row is essentially optimal.

BTRAN by FTRAN on the transpose Following [62], the transpose of the LU factors of the basis matrix is found, and FTRAN with the transposed factors is preferred to BTRAN on the originals. Etas cannot be skipped in BTRAN, so that performing it as FTRAN instead is dramatically more efficient for many problems. For update etas, BTRAN is still performed, but these are comparatively few in number.

Iterative refinement This is a well-known technique whereby the residuals after a solve with FTRAN or BTRAN are calculated, and repeat solves are made with them to improve the accuracy of the resulting vector. This can be used to eliminate suspected perturbed zeroes, and also to improve the accuracy of any solution the code returns.

Structurisation by hypergraph partitioning Following [9], unstructured problems may be converted, after they are read, into primal block angular form, using PATOH [27] for hypergraph partitioning. This method is a substantial improvement upon the bipartite graph partitioning approach [38], which is also available, and makes use of METIS [73].

Maintenance of $\hat{R}_i$ To reduce the number of solves with T, $\hat{R}_i = T_i^{-1} R_i$ are maintained as thin, sparse tableaux.

A note on the product form

`i7` relies on the elimination form of the inverse with product form updates [89, 30] for its matrices T_i which is an obsolete technique. Alternatives based on updating the LU factors directly [11, 42, 102, 111] are well known, and were implemented separately as part of `i5` (see Appendix D). The decision to use the product form can only be attributed to ignorance: performance studies of simplex technologies are seldom published, and it was believed that a fully optimized solver based on this update could be competitive for general problems. As the results will make clear, this is unlikely to be the case.

3.7.2 Parallel techniques

Task farm job management This is a standard paradigm for the construction of parallel programs. Parallel infrastructure is isolated in the code by making all parallelised tasks schedule jobs to a queue. This allows a wide range of routines to be accommodated without excessive code complexity.

Fork-join parallelism Sections of code are either parallel, so that all threads execute symmetrically on chunks of a task, or serial, with work performed by a single thread. Each parallel region begins and ends with a full synchronisation barrier, so that parallel sections do not overlap.

Lock-free datastructures Where contention between different workers for data is unavoidable, for example the job queue, lock-free structures are used where possible. These rely on processor-level atomic instructions, and impose negligible overheads.

Spin-locks When lock-free structures cannot easily be used, spin locks are preferred to mutexes. Here, a worker which wishes to obtain a lock monitors a memory location in a tight loop ("spins"). Their principle advantage is that the thread need not enter kernel space, especially when access is expected to be granted after a short delay.

Spin-barriers Barrier synchronisations occur after each parallel section, and there are several per iteration, so that the interval between them is very short. A typical POSIX barrier generates a thread yield, which can lead to poor scheduling decisions on the part of the operating system. To improve parallel performance, a global sense-reversing barrier [94] was implemented in assembly language.

Deterministic parallelism Given the same options, the solve for a given problem will retrace an identical path every time it is run. Additionally, this path is constant across different numbers of workers, provided the number and location of the blocks identified in the constraint matrix remains constant.

Segmented indexed vectors Sparse vectors are divided into segments, with separate lists of nonzeroes in each segment, so that workers can perform sparse operations without contending to modify the set of nonzero positions.

Superblocking To prevent excessive scheduling overhead on solves with many blocks and few workers, many small blocks can be grouped together into a *superblock* which is scheduled as a single unit.

3.7.3 Parallel elements

The following routines in `i7` have one or more parallel parts:

FTRAN FTRAN on input vectors matching the block structure performs the multiplications with $\hat{R}$ in parallel. For unstructured vectors, the solves with T are also done in parallel.

BTRAN BTRAN on input vectors matching the block structure performs the solves with $T^\top$ in parallel. For unstructured vectors, the multiplications with $\hat{R}$ are also parallelised.

Invert The factorisation of each T is performed in parallel, as is the computation of each $\hat{R}$ tableau. The Schur complement W is regenerated in stripes in parallel, then aggregated and factorised in serial.

Tableau row generation The calculation of the pivot row is performed in parallel.

Column selection The traversal to find the most attractive entering column during full pricing is done in parallel, as is the update of the column norms. In list pricing, the update to the stored columns is performed in parallel.

Pivot selection The accumulation of breakpoints in phase one is performed in parallel, as is the scan for the minimal element in the phase two ratio test. In Wolfe's method, the virtual perturbations are applied, removed and updated in parallel.

Shift The update of the primal values is performed in parallel.

3.8 Results

This section presents performance analyses for `i7` on a small selection of real-world problems. Results for a number of other test sets can be found in Appendix E, which includes data for multi-commodity flow problems, as well as for the Netlib set, the latter being a demonstration of the code's numerical stability.

3.8.1 Baseline performance

Any results on the parallelisation of the simplex method can be of only limited interest when the serial performance of the same code is hopelessly uncompetitive. The most obvious approaches to accelerating the non-trivial parts of the simplex method involve the parallelisation of inefficiencies which ought instead to be removed from a practical code.

Problem			`i7`	`glpk`	`clp`	`Xpress`
Name	Rows	Columns	Time (s)	Time (s)	Time (s)	Time (s)
DANO3MIP_LP	3202	13873	116.53	10.8	14.01	11.12
GEN4	1537	4297	159.59	50.4	44.12	37.17
LP22	2958	13434	742.89	54.0	32.58	26.82
PDS-40	66844	212859	283.09	2073.3	161.41	191.77
RLFPRIM	58866	8052	79.54	8.9	4.77	2.28
STORMG2-27	14441	34114	2.38	17.1	4.28	0.88
STORMG2-125	66185	157496	55.80	490.8	121.42	18.76
NUG08	912	1632	5.86	0.8	0.74	1.66
NSCT2	23003	14981	3.61	9.2	1.75	3.62
SGPF5Y6	246077	308634	287.51	1820.6	82.32	8.08
WORLD	35510	32734	612.92	387.3	178.11	370.69
DCP1	4950	3007	1.14		1.61	1.18

Larger test problem results on `richtmyer` from § E.3. Continued on the next page.

Problem			i7	glpk	clp	Xpress
Name	Rows	Columns	Time (s)	Time (s)	Time (s)	Time (s)
DCP2	32388	21087	81.35		70.03	57.02
DETEQ8	20678	56227	16.68	34.4	9.27	0.96
DETEQ27	68672	186928	174.58	397.5	100.04	6.46
		Geometric mean	51.06	64.33	19.07	9.67

Larger test problem results on `richtmyer` *from § E.3.*

`i7` shows good overall performance on `STORMG2-125` and `DCP1`, but relatively poor performance for a number of other instances. Overall, it is faster on this set than `glpk`, which also fails to solve two instances, but slower than `clp`, the leading open-source code, and also slower than `Xpress`, a leading commercial code. The disparity is not so great as to make any comparisons meaningless however.

Problem			i7	glpk	clp	Xpress
Name	Rows	Columns	Iter/sec	Iter/sec	Iter/sec	Iter/sec
DANO3MIP_LP	3202	13873	555.86	1293.70	1120.68	4921.49
GEN4	1537	4297	79.86	183.71	288.45	137.94
LP22	2958	13434	132.71	751.56	782.76	1362.10
PDS-40	66844	212859	608.11	208.77	877.49	4124.96
RLFPRIM	58866	8052	124.06	600.45	1508.59	4668.20
STORMG2-27	14441	34114	8096.86	1371.35	4656.47	26581.35
STORMG2-125	66185	157496	1626.96	282.33	911.38	6340.46
NUG08	912	1632	1220.50	4186.25	3871.97	3865.02
NSCT2	23003	14981	4721.64	1351.30	7034.82	4906.07
SGPF5Y6	246077	308634	861.74	129.33	3164.50	30289.69
WORLD	35510	32734	204.79	389.57	572.08	721.88
DCP1	4950	3007	5025.37	7470.00	3026.05	6256.83
DCP2	32388	21087	676.66	1152.12	547.95	1425.80
DETEQ8	20678	56227	2521.07	936.83	3499.35	35074.33
DETEQ27	68672	186928	840.28	279.52	1145.87	17667.21
		Geometric mean	810.63	707.48	1506.25	4527.23

Larger test problem results on `richtmyer` *from § E.3.*

A comparison of iteration rates is difficult to make reasonably. To some extent, the iteration rate of a primal simplex code is a design decision, where more or fewer iterations can be traded off against faster cycles. The effect of the sparsity of the basis factorization, and of the particular path the solver follows, can also be pronounced. The strategy implemented in `i7` is a different compromise from those in the other solvers, as there is no partial pricing, only a fast, weak normalised pricing that is constantly applied. This is reflected to some extent in the iteration rates, though many other factors are involved.

3.8.2 Structurisation

This test set contains general, real-world problems which are not marked previously with their structure. If it were to prove very expensive to obtain this structure, then the method itself might be of little interest, at least when applied automatically to general problems.

Problem			2 blocks	4 blocks	8 blocks	16 blocks
Name	Rows	Columns	Time (s)	Time (s)	Time (s)	Time (s)
DANO3MIP_LP	3202	13873	0.12	0.13	0.14	0.16
GEN4	1537	4297	0.06	0.07	0.08	0.09

Larger test problem results on `richtmyer` *from § E.3. Continued on the next page.*

Problem			2 blocks	4 blocks	8 blocks	16 blocks
Name	Rows	Columns	Time (s)	Time (s)	Time (s)	Time (s)
LP22	2958	13434	0.05	0.06	0.09	0.10
PDS-40	66844	212859	0.43	0.62	0.79	0.94
RLFPRIM	58866	8052	0.18	0.23	0.29	0.34
STORMG2-27	14441	34114	0.06	0.08	0.11	0.14
STORMG2-125	66185	157496	0.27	0.37	0.49	0.61
NUG08	912	1632	0.01	0.01	0.01	0.01
NSCT2	23003	14981	0.55	0.74	0.92	0.98
SGPF5Y6	246077	308634	0.77	1.03	1.37	1.65
WORLD	35510	32734	0.14	0.19	0.23	0.28
DCP1	4950	3007	0.03	0.04	0.05	0.06
DCP2	32388	21087	0.16	0.26	0.34	0.41
DETEQ8	20678	56227	0.09	0.13	0.17	0.21
DETEQ27	68672	186928	0.29	0.41	0.54	0.66
	Geometric mean		0.12	0.17	0.21	0.25

Larger test problem results on `richtmyer` *from § E.3.*

As can be seen, the cost of obtaining a given structure for these problems using the method of Aykanat et al. [9] is relatively minor, amounting to only a small fraction of the time required to solve the problem.

Problem			2 blocks Linking	4 blocks Linking	8 blocks Linking	16 blocks Linking
Name	Rows	Columns	rows (%)	rows (%)	rows (%)	rows (%)
DANO3MIP_LP	3202	13873	1277 (40)	1471 (46)	1611 (50)	1764 (55)
GEN4	1537	4297	1019 (66)	1038 (68)	1409 (92)	1442 (94)
LP22	2958	13434	683 (23)	697 (24)	714 (24)	838 (28)
PDS-40	66844	212859	1516 (2)	4312 (6)	5232 (8)	6223 (9)
RLFPRIM	58866	8052	3386 (6)	6322 (11)	11163 (19)	14382 (24)
STORMG2-27	14441	34114	293 (2)	684 (5)	1013 (7)	1313 (9)
STORMG2-125	66185	157496	1341 (2)	2909 (4)	4346 (7)	5613 (8)
NUG08	912	1632	470 (52)	595 (65)	629 (69)	658 (72)
NSCT2	23003	14981	1786 (8)	2193 (10)	3421 (15)	3802 (17)
SGPF5Y6	246077	308634	675 (0)	890 (0)	2902 (1)	3650 (1)
WORLD	35510	32734	265 (1)	545 (2)	991 (3)	2106 (6)
DCP1	4950	3007	39 (1)	39 (1)	79 (2)	120 (2)
DCP2	32388	21087	43 (0)	68 (0)	233 (1)	295 (1)
DETEQ8	20678	56227	327 (2)	535 (3)	686 (3)	1417 (7)
DETEQ27	68672	186928	965 (1)	1591 (2)	2030 (3)	2373 (3)
	Geometric mean		3%	5%	8%	10%

Larger test problem results on `richtmyer` *from § E.3.*

The quality of the obtained structure varies considerably on this set. The problems for which the structure is of the highest quality, `STORMG2-125` and `DCP2`, are known to be naturally primal block angular, although the code is not aware of this fact. `DANO3MIP` and `GEN4` are examples of a problem in which structure cannot be easily found, and indeed the requested number of blocks cannot always be found in either.

3.8.3 Structured performance

Having established that the serial code is sufficiently developed to sustain an analysis of parallel performance, the next question should be to what degree inefficiencies inherent in operating with the structured basis matrix can be expected to limit practical performance. The following comparisons are of serial performance.

Problem			2 blocks	4 blocks	8 blocks	16 blocks
Name	Rows	Columns	Time cost	Time cost	Time cost	Time cost
DANO3MIP_LP	3202	13873	1.65	1.46	1.22	1.24
GEN4	1537	4297	0.69	0.44	0.65	0.66
LP22	2958	13434	0.46	0.71	0.36	1.63
PDS-40	66844	212859	2.00	3.21	2.23	1.52
RLFPRIM	58866	8052	0.17	0.16	0.28	0.24
STORMG2-27	14441	34114	1.57	1.55	1.50	1.57
STORMG2-125	66185	157496	1.63	1.74	1.43	1.30
NUG08	912	1632	0.89	0.74	0.89	1.05
NSCT2	23003	14981	2.47	1.87	2.41	2.31
SGPF5Y6	246077	308634	0.49	0.42	0.75	0.61
WORLD	35510	32734	0.91	0.75	0.53	0.58
DCP1	4950	3007	0.95	0.87	0.67	0.80
DCP2	32388	21087	1.84	1.10	0.81	0.71
DETEQ8	20678	56227	1.29	1.19	1.15	1.19
DETEQ27	68672	186928	1.27	1.03	0.85	0.76
	Geometric mean		1.02	0.93	0.89	0.95

Larger test problem results on `richtmyer` *from § E.3.*

The time cost of the structured form is the ratio of the run-time of the structured code to the unstructured code, in serial. Thus, in fact, the structured form is on average more efficient than the unstructured form for `i7` on this test set. These gross statistics hide considerable variability however. Particularly notable is the high cost of the structured form on `PDS-40`, despite the good quality of the structured form found, and the very low cost of `GEN4` despite the extremely poor structured form found.

Problem			2 blocks	4 blocks	8 blocks	16 blocks
Name	Rows	Columns	Rate cost	Rate cost	Rate cost	Rate cost
DANO3MIP_LP	3202	13873	1.78	1.48	1.35	1.25
GEN4	1537	4297	0.29	0.47	0.72	0.78
LP22	2958	13434	0.37	0.34	0.23	0.29
PDS-40	66844	212859	1.73	2.97	1.84	1.34
RLFPRIM	58866	8052	0.23	0.23	0.33	0.33
STORMG2-27	14441	34114	1.50	1.56	1.50	1.52
STORMG2-125	66185	157496	1.64	1.73	1.46	1.34
NUG08	912	1632	0.73	0.80	0.81	0.86
NSCT2	23003	14981	2.48	1.82	2.30	2.17
SGPF5Y6	246077	308634	0.49	0.42	0.74	0.61
WORLD	35510	32734	0.92	0.63	0.51	0.52
DCP1	4950	3007	1.02	0.90	0.72	0.82
DCP2	32388	21087	1.90	1.12	0.82	0.72
DETEQ8	20678	56227	1.32	1.20	1.12	1.18
DETEQ27	68672	186928	1.32	1.06	0.87	0.80
	Geometric mean		0.96	0.90	0.87	0.85

Larger test problem results on `richtmyer` *from § E.3.*

The rate cost of the structured form is the ratio of the iteration rate of the unstructured form to the structured form. Once again, the structured form is on average more efficient, and by approximately the same amount. This would be expected since there are no inherent alterations to the pricing or pivoting rules when switching between structured and unstructured systems.

3.8.4 Parallel performance

It is apparent that the performance of `i7` when operating with the structured form is not substantially different on this test set from its performance when using the unstructured form. The factors which will limit parallel performance are therefore three-fold: the overhead of contention for resources; the proportion of the code which is in fact executed in parallel (Amdahl's law); and finally the quality of the load-balancing between the processors.

Problem			2 block, 2 core speedup	4 block, 4 core speedup	8 block, 8 core speedup	16 block, 16 core speedup
Name	Rows	Columns				
DANO3MIP_LP	3202	13873	0.96	0.99	1.01	1.02
GEN4	1537	4297	1.10	1.11	1.00	1.00
LP22	2958	13434	1.08	1.05	1.20	1.10
PDS-40	66844	212859	1.18	1.84	2.06	1.76
RLFPRIM	58866	8052	1.05	0.88	1.23	1.18
STORMG2-27	14441	34114	0.93	1.20	1.19	1.06
STORMG2-125	66185	157496	1.05	1.35	1.53	1.69
NUG08	912	1632	0.95	0.98	0.93	0.94
NSCT2	23003	14981	1.24	1.12	1.29	1.20
SGPF5Y6	246077	308634	1.05	1.23	1.80	1.80
WORLD	35510	32734	1.29	1.58	1.51	1.45
DCP1	4950	3007	0.99	1.07	1.01	0.94
DCP2	32388	21087	1.55	1.79	1.69	1.55
DETEQ8	20678	56227	0.99	1.15	1.22	1.12
DETEQ27	68672	186928	1.10	1.33	1.52	1.64
	Geometric mean		1.09	1.22	1.31	1.26

Larger test problem results on `richtmyer` *from § E.3.*

The speed-up given here is over the structured code running in serial on the same number of blocks. As previously noted, the structured form is on average more efficient than the unstructured, which makes these numbers a conservative (but entirely more meaningful) estimate of speed-up. Speed-up is bounded at a very low multiple for any number of cores, and indeed above 8 cores, performance begins to degrade. To what extent this effect is driven by thermal effects in the processor packages as opposed to load balancing and contention is unclear.

Problem			2 block, 2 core Peak	4 block, 4 core Peak	8 block, 8 core Peak	16 block, 16 core Peak
Name	Rows	Columns				
DANO3MIP_LP	3202	13873	0.94	0.94	0.92	1.04
GEN4	1537	4297	1.33	1.45	1.13	1.03
LP22	2958	13434	1.10	1.03	1.21	1.33
PDS-40	66844	212859	1.23	2.09	2.74	2.22
RLFPRIM	58866	8052	1.14	0.72	2.09	1.95
STORMG2-27	14441	34114	0.93	1.29	1.18	1.07
STORMG2-125	66185	157496	1.05	1.64	1.96	2.23
NUG08	912	1632	0.82	0.80	0.76	0.56
NSCT2	23003	14981	1.25	1.28	1.99	1.38
SGPF5Y6	246077	308634	1.05	1.26	2.11	2.04
WORLD	35510	32734	1.35	1.72	1.87	2.03
DCP1	4950	3007	0.98	1.04	0.91	0.77
DCP2	32388	21087	1.61	1.86	2.03	1.88
DETEQ8	20678	56227	1.04	1.20	1.61	1.23
DETEQ27	68672	186928	1.16	1.65	2.22	2.11
	Geometric mean		1.12	1.28	1.54	1.41

Larger test problem results on `richtmyer` *from § E.3.*

The peak speed-up given here is the amount that the actually parallel part of the computations was accelerated by. This is calculated by measuring the serial parts of the solve time on a parallel run, and deducting that amount also from the serial solve, giving an approximate peak speed-up measurement. Once again, speed-up is bounded at a low level, mostly below 2×, although there are exceptions which near 3.0×. Thus, even the code which is parallelised does not speed-up by much more than twice, with any number of cores.

Problem			16 block, 2 core speedup	16 block, 4 core speedup	16 block, 8 core speedup	16 block, 16 core speedup
Name	Rows	Columns				
DANO3MIP_LP	3202	13873	0.94	0.98	0.97	1.02
GEN4	1537	4297	1.00	1.00	1.01	1.00
LP22	2958	13434	0.97	1.08	1.10	1.10
PDS-40	66844	212859	1.20	1.55	1.77	1.76
RLFPRIM	58866	8052	1.04	1.14	1.17	1.18
STORMG2-27	14441	34114	0.93	1.00	1.16	1.06
STORMG2-125	66185	157496	1.16	1.46	1.59	1.69
NUG08	912	1632	0.87	0.93	0.95	0.94
NSCT2	23003	14981	1.06	1.21	1.41	1.20
SGPF5Y6	246077	308634	1.27	1.53	1.75	1.80
WORLD	35510	32734	1.08	1.31	1.41	1.45
DCP1	4950	3007	0.76	0.88	0.90	0.94
DCP2	32388	21087	0.99	1.32	1.56	1.55
DETEQ8	20678	56227	0.95	1.14	1.20	1.12
DETEQ27	68672	186928	1.11	1.40	1.59	1.64
	Geometric mean		1.02	1.17	1.27	1.26

Larger test problem results on `richtmyer` *from § E.3.*

The final comparison is between different numbers of threads on the same block structure, chosen here to be 16 for generality. If load balancing were to be responsible for the failure of the parallel code to accelerate, then we might expect different behaviour with more blocks than threads. However, this does not occur, as can be seen, and the speed-up numbers are almost unchanged from the block per core results given earlier.

3.9 Conclusions

Kaul's method can be implemented in a practical code, and can solve real problems with no less accuracy than is entailed by the use of a traditional simplex basis. This chapter has presented a working proof of this fact; indeed `i7` may be the first ever working implementation of this method.

However, parallel implementations of the simplex method are memory bound on multi-core hardware, which was a conclusion of Chapter 2 and is also a conclusion of this one. With two processor packages, speed-up of around two is possible when applying increasing numbers of workers to a structured representation, but the additional overhead of this representation decreases the practical speed-up achieved considerably. Only for a handful of problems does this approach appear worthwhile, and the extent of path effects on these cases cannot be certainly quantified.

Against these marginal gains in performance must be weighed the substantial effort and complexity involved in creating a simplex solver of this type. Parallelism creates numerous problems of reproducibility and portability, and

the structured form requires extensive code to support it, over and above that required for a revised simplex solver. Should commodity hardware evolve to the point at which the memory bandwidth exists to subserve parallel computation in sparse arithmetic, this method may become attractive. When the speed-up is strictly bounded at a small fraction of the cores involved, however, it would be difficult to justify the implementation effort.

Chapter 4

hmf - Parallel matrix-free interior point

4.1 Introduction

Parallel implementations of interior point methods for linear programming have been reported by several authors [e.g. 16, 71, 8], and have been shown to be both efficient and also capable of excellent speed-up. At the heart of such a code is a factorisation step, which is repeated at each iteration, and for which advanced, parallel techniques are available [67]. This makes a significant contribution to the overall speed-up achieved by such solvers.

In a *matrix-free* interior point solver [51], direct factorisation is replaced by the use of iterative methods, and although such approaches are experimental at this time, they offer the prospect of solving efficiently some classes of problem which are presently extremely difficult. A significant part of the run-time of such a code is spent in performing sparse matrix-vector products, and it may be hoped that a parallel implementation of these routines will provide useful speed-up for the entire solver.

This chapter describes `hmf`, an application of the matrix-free Hopdm interior point library [7, 49], which can exploit both multi-core and many-core hardware. For two difficult problem classes, `hmf` is shown to produce results with exceptional speed, although the accuracy of the solutions obtained is low. A novel approach for performing sparse matrix-vector products on a GPU is also described, and is shown to improve the performance of multiplications with the constraint matrix for these problems. This work previously appeared in [109].

4.2 Matrix-free interior point methods

The major parts of interior point methods were covered briefly in § 1.3.4, where the necessity of calculating a search direction $(\Delta x, \Delta y, \Delta s)$ at each iteration was described. This direction is found by solution of the system

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I_n \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^T y - s \\ \mu e - XSe \end{bmatrix}, \tag{4.1}$$

in which the terms X and S on the right-hand side vary between iterations.

This form is usually reduced to the *normal equations* [5], given by

$$A\Theta A^\top \Delta y = g, \tag{4.2}$$

in which $\Theta = XS^{-1}$ is a diagonal matrix,

$$g = A\Theta A^\top y - A\Theta\left(c - \mu X^{-1}e\right) + Ax - b, \tag{4.3}$$

and $(\Delta x, \Delta s)$ follow by substitution. These have the advantage of requiring only the factorisation of a symmetric, positive definite matrix $G = A\Theta A^\top$. When this is treated by Cholesky decomposition [14], giving $G = LDL^\top$, the sparsity pattern of G and L remains constant between iterations, and this enables much of the analysis required for an efficient factorisation to be re-used.

There exist sparse problems, however, for which the factor L is near full, making it very expensive to compute or store. To avoid forming L, matrix-free methods solve the normal equations iteratively, for example by using conjugate gradients [64]. This will typically require many multiplications with G but, when A is sparse, may still be comparatively attractive. The following discussion is based on [51], which goes into much more detail.

The rate of convergence of conjugate gradients depends upon the ratio of the largest to the smallest eigenvalue in the target matrix. In the case of the normal equations, note that Θ is a diagonal matrix whose elements are the ratios x_i/s_i, one or other of which must tend to zero at the optimum. Intuitively, the eigenvalues of G will be dispersed by Θ as the solve progresses, and when conjugate gradients are applied to it directly, convergence will suffer.

This tendency can be counteracted by *regularising* the original problems. A quadratic term, derived from the distance between the current solution and fixed reference points $\bar{x}$ and $\bar{y}$, is added to the objective, giving the problems

$$\begin{aligned} \text{minimise} \quad & c^\top x + r^{(p)} \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \\ \text{where} \quad & r_i^{(p)} = \frac{p_i}{2}(x_i - \bar{x}_i)^2, \end{aligned} \tag{4.4}$$

$$\begin{aligned} \text{maximise} \quad & b^\top y + r^{(d)} \\ \text{subject to} \quad & A^\top y + s = c \\ & s \geq 0 \\ \text{where} \quad & r_i^{(d)} = \frac{d_i}{2}(y_i - \bar{y}_i)^2. \end{aligned} \tag{4.5}$$

The rôle of p and d becomes immediately clear in the normal equations for the corresponding barrier problems,

$$\left(A\left(\Theta^{-1} + P\right)^{-1} A^\top + D\right)\Delta y = g, \tag{4.6}$$

in which P provides an upper bound on the magnitude of the largest eigenvalue, and D a lower bound upon the magnitude of the smallest.

Convergence can be further improved by the use of a preconditioner. Consider performing a Cholesky factorisation of G, then at an intermediate point, this calculation can be written

$$G = \begin{bmatrix} L_u & \\ L_\ell & I \end{bmatrix} \begin{bmatrix} V_u & \\ & G' \end{bmatrix} \begin{bmatrix} L_u^\top & L_\ell^\top \\ & I \end{bmatrix}, \tag{4.7}$$

where V_u is diagonal, and G' is the active submatrix. This suggests a preconditioner Ψ, requiring only the the diagonal $V_{G'}$ of G', of the form

$$\Psi = \begin{bmatrix} L_u & \\ L_\ell & I \end{bmatrix} \begin{bmatrix} V_u & \\ & V_{G'} \end{bmatrix} \begin{bmatrix} L_u^\top & L_\ell^\top \\ & I \end{bmatrix}. \tag{4.8}$$

4.3 Sparse matrix-vector products

A matrix-free interior point solver devotes much of its run-time to the multiplication of vectors by the constraint matrix A, and to operations with the previously described preconditioner Ψ. Only the former will be considered in this work, which is to say that strategies will be described for multiplying a dense vector by a sparse matrix A or its transpose, and also for calculating the compound product $A\Theta A^\top x$. For many-core hardware, such techniques are in their infancy, and several of the methods detailed in § 4.3.2 could not be uncovered in earlier literature.

4.3.1 On multi-core hardware

Whilst performing sparse arithmetic, a processor accesses memory at locations determined by the pattern of nonzeroes in the working vector and matrix, and these patterns are not easily predicted. Moreover, in the calculation of sparse matrix-vector products, there is little opportunity for data re-use, as the matrix is read only once, and for this read to be efficient, scattered accesses must be made to the working vector.

Current generation hardware attempts to compensate for the high latency and low bandwidth of main memory, but the previous considerations make the methods employed largely ineffective for this workflow. It must be expected, therefore, that the rate at which such computations can be completed is bounded by the performance of the memory subsystem, and that optimizations to reduce instruction overhead and improve throughput will have limited effect. For vectorization in particular, nonzeroes must be gathered into adjacent locations, and this may lead to a net degradation in performance.

Reducing memory traffic provides more substantial gains. Vuduc et al. [119] describe the following trick, whereby for A held column-wise in

$$y = A\Theta A^\top x, \tag{4.9}$$

all operations can be completed with a particular column a_i before the next column is read, using

$$y = A\Theta A^\top x = \sum_i \theta_i (a_i^\top x) a_i. \tag{4.10}$$

Columns sufficiently small enough to be contained in cache will now be read from memory only once.

Despite this technique, and despite the fact that all of the target operations are embarrassingly parallel, or nearly so, limited speed-up can be expected due to memory contention. This was also seen for tableau updates in Chapter 2.

4.3.2 On many-core hardware

A GPU also achieves its peak performance for dense arithmetic, and the reduction in the sparse case is even more pronounced, as such devices support only certain patterns of memory accesses efficiently (see § 1.2.5). This makes the choice of matrix representation a critical factor in determining the speed of kernels for sparse matrix-vector multiplications on this platform, and the optimal such selection is problem dependent [12].

A non-exhaustive list of representations for sparse matrices on such devices is now provided. The TELL representation, and the dense-hybrid representations, were previously described by the author in [109], but are not otherwise known to have been published.

Compressed sparse row (CSR) [12] The rows of a matrix are stored end-to-end in two memory blocks, one containing nonzero matrix entries, and the other containing the column indices of those entries. This requires $O(z)$ memory locations, for z the number of nonzeroes in the matrix. Coalesced access to this data can be ensured by applying an entire warp of threads (32) to each row.

ELLPACK (ELL) [12, 118] The rows of a matrix are stored as two lists of vectors, one list containing matrix entries, and the other containing column indices [57]. Each vector has one entry per row, so that elements from different rows are interleaved in memory, making all accesses coalesced at one thread per row. Entries for a particular row occur with a fixed stride, and all rows are padded to the same length, which requires $O(\ell m)$ memory locations, where ℓ is the length of the longest row and m is the number of rows.

Coördinate (COO) [12] There are three memory blocks, one for row indices, one for column indices, and one for matrix entries, requiring $O(z)$ storage. Coalesced access is possible, but effective routines making use of this representation are complex, and have low performance for many matrices.

ELLPACK-COO hybrid format (HYB) [12] The bulk of the matrix is stored in ELL format, with nonzeroes from long rows placed into a COO section. This yields a format which is practical for general matrices.

ELLPACK-R [118] This is the ELL representation with the addition of a list of row lengths, which allows operations with zero padding to be avoided. Zero padding must, however, still be stored.

Transposed ELL (TELL) The ELL format may be transposed yielding a CSR-like representation with fixed-length rows and controlled alignment. As for CSR, coalesced access can be ensured with one warp per row or, on newer devices, one half-warp per row.

Dense-hybrid ELL and TELL (DHELL and DHTELL) These formats store a portion of the rows of the matrix as dense, say those with fewer than 25% of entries at zero, and the remainder of the matrix as either ELL or TELL respectively. They are appropriate for matrices with a small, dense section whose other rows are of a similar length.

Different numbers of threads may be applied to the operations with each row of these representations, as identified by Bell and Garland [12] for CSR, and Vázquez et al. [118] for ELL-R. Coalesced access can be maintained for between 1 and 16 threads with ELL, and for between 32 and 512 threads with TELL and CSR. On newer hardware, these rise to between 1 and 32 threads for ELL, and to between 16 and 512 threads for TELL.

4.4 Implicit constraints

When direct factorisation of G in the normal equations is no longer performed, the amount of data which must be processed explicitly at each iteration is significantly reduced. Further gains result from the observation that access to the constraint matrix occurs only in the context of a small number of well-defined operations, and that these operations could be as easily provided by black box functions, which is to say an *oracle*, as by naïve calculation with A. This section describes the creation of such an oracle for the linear relaxations of quadratic assignment problems. A previous description of the use of an oracle for a matrix-free interior point solver could not be identified in the literature.

Quadratic assignment problems arise, for example, when a set of facilities must be placed so as to minimise the cost of transportation between them. The task in general is to find a minimising permutation φ in

$$\text{minimise} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij}\, d_{\varphi(i)\varphi(j)} + \sum_{i} b_{i\varphi(i)}, \tag{4.11}$$

representing, for example, a set of n facilities, with flows f_{ij} between them, which must be placed on a set of n points, separated by distances d_{ij}, with fixed cost b_{ij} to place facility i at point j. Such problems can be difficult, and a number of standard instances are unsolved at this time [101].

The Adams-Johnson linearisation [4, 23] is an equivalent formulation without quadratic terms. Its closed form is

$$\begin{aligned}
\text{minimise} \quad & \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} c_{ijkl}\, y_{ijkl} \\
\text{subject to} \quad & \sum_{i=1}^{n} y_{ijkl} = x_{kl} \qquad \forall j,k,l \in N, \qquad \sum_{j=1}^{n} y_{ijkl} = x_{kl} \qquad \forall i,k,l \in N, \\
& \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in N, \qquad \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in N, \\
& y_{ijkl} = y_{klij} \qquad \forall i,j,k,l \in N, \qquad y_{ijkl} \geq 0 \qquad \forall i,j,k,l \in N, \\
\text{where} \quad & x_{ij} \in \mathbb{B}, \;\; y_{ijkl} \in \mathbb{R}.
\end{aligned} \tag{4.12}$$

Here $\mathbb{B} = \{0, 1\}$, and when the requirement for integrality is relaxed, the result is a continuous program, suitable for treatment by interior point methods, which can be used to bound the optimal objective. Reductions in this form for symmetric problems are possible, but the size of the relaxation grows in any case as $O(n^4)$ in the original parameter n, so that by $n = 80$, the constraint matrix for even a symmetric problem is already of dimension $1,011,360 \times 19,977,600$.

The constraint matrix of the relaxation (4.12) is fixed for all quadratic assignment problems of equal size, with only the objective varying between instances, and furthermore all coefficients in this matrix are unitary. When a closed form generator for rows and columns of such a matrix is known, it may be specialised in a straightforward manner to perform matrix-vector multiplication during its traversal, rather than writing out coefficients, and this provides the core of an oracle for matrix-free interior point methods on such problems.

4.5 Parallel matrix-free interior point

`hmf` is based on the matrix-free version of the HOPDM library for solution of linear programs by interior point methods [7, 49]. It consists of a front-end code, capable of reading both linear programs and quadratic assignment programs, and back-end routines to perform sparse matrix-vector multiplication with the constraint matrices of these problems.

The back-end is organised into modules, each of which provides routines to perform the sparse matrix-vector products $y = Ax$ (FSAX), $x = A^\top y$ (FSATY) and $x = A\Theta A^\top y$ (FSAAT). In addition to an optimized serial module, there is a parallel multi-core implementation using POSIX thread primitives, and a GPU module for NVIDIA devices. A further, serial, module can function as an oracle for quadratic assignment programs in this setting, by providing both the previous operations and also, from $A\Theta A^\top$, particular columns (SCLAAT), the diagonal (SDGAAT), and density estimates (MFCAAT).

The GPU module provides access to several CUDA kernels for operating on the CSR, DHELL and DHTELL representations, each of which applies different numbers of threads to calculations with rows of the constraint matrix. The matrix and its transpose are stored separately using a given representation, and $A\Theta A^\top$ is calculated naïvely by the application of three kernels, the middle of which performs a trivial multiplication with Θ.

The oracle implementation is specialised for symmetric quadratic assignment problems, and stores just the $n \times n$ matrices of the original problem definition. Only the routine MFCAAT generates any part of the constraint matrix explicitly, and this is called only once, during the creation of the preconditioner, to identify sparse columns to bring into the partial Cholesky factorisation. All other operations are implemented implicitly, by specialising a linear program generator for such problems so that it transforms an input vector using the components of the matrix it would have generated. The linear program generator from which this is derived represents a complete rewrite of the `newlp` program [68] to enable generation of individual rows and columns in memory and on demand. Only constant-sized working space is required above that used for the vectors involved.

4.6 Results

This section first presents results showing the performance of GPU kernels for sparse matrix-vector multiplication, operating on the specialised representations described in § 4.3.2, and making use of varying numbers of threads per row. These are followed by results for `hmf` on a test-set of quadratic assignment problems, for which both measures of accuracy and time to solution are given. The final part of this section describes the performance of `hmf` on non-classicality threshold subproblems for multiqubit states [58], which are another set of linear programming formulations difficult to solve by traditional means.

All results for `hmf` were obtained on the machine `grunty`, described in § 1.2.6. A standard benchmark of numerical stability is not provided, as `hmf` is dependent on problem specific tuning, and not capable of the solution of general problems. The ongoing problem of accuracy in this framework will be discussed alongside the relevant results.

4.6.1 Sparse matrix-vector kernels

As shown by Bell and Garland [12], the performance of kernels for sparse matrix-vector multiplication depends in large part upon the characteristics of the matrices to which they are applied. Table 4.1 shows the test instances used in this section, which are the constraint matrices of problems investigated in subsequent parts. The first four instances are from the quantum theory subproblems, provided by Gruca et al. [58], and are both large and relatively dense, being notable for containing a single completely dense row and column. The remaining instances are from the linearisations of quadratic assignment problems, and are considerably larger, but also more sparse.

Matrix	Rows	Columns	Nonzeroes
16Kx16K0	16,385	16,385	2,129,920
64Kx64K0	65,537	65,537	16,908,288
96x128-0	98,305	131,073	50,561,024
256x256-0	262,145	262,145	134,742,016
esc16a	7,712	29,056	123,392
esc32a	63,552	493,056	2,033,664
esc64a	516,224	8,132,608	33,038,366
tai80a	1,011,360	19,977,600	80,908,800

Table 4.1: Dimensions of test matrices.

The kernels for the CSR representation using 1 or 32 threads per row were described previously by Bell and Garland [12], and the tested implementations are largely equivalent. The remaining kernels vary in their novelty. Results for CSR with this range of threads per row could not be found in the literature. Although Vázquez et al. [118] described kernels for the ELL-R representation with 1 and 8 threads per row, results using the dense-hybrid representation, or with more than 8 threads per row, have not been discovered in previous work. The TELL representation is believed novel, and hence no kernels using its dense-hybrid representation are known to have been previously analysed.

The results give the time required to perform a single multiplication of random dense vector with a sparse matrix, averaged over five attempts. For each problem, the fastest average time is marked. Overall, the fastest kernel is that which applies one half-warp, or 16 threads, to each row of DHTELL representations of these matrices. The performance of CSR with one thread per row, which is the only kernel having uncoalesced memory reads, illustrates the importance of ensuring proper access patterns on this platform.

4.6.2 Quadratic assignment problems

In this section, the performance of general purpose, accelerated linear algebra for matrix-free interior point methods is evaluated on quadratic assignment problems. The performance of an oracle for the constraint matrices of such problems is also demonstrated. The instances used for testing in this section are all symmetric, and drawn from a standard test-set [101]. The first family of problems, `esc16a` and its larger relatives, originate from minimising the amount of additional hardware required to make sequential circuits self-testable [37],

and have known optima. The remainder, TAI10A and its relatives, constitute a family of uniformly generated problems first proposed by Taillard [114], and the selected instances TAI40A, TAI50A, TAI60A and TAI80A are unsolved as of April 2012.

The first result table presents the time to solution for each linear relaxation using each of hmf's four linear algebra modules, serial code, parallel multi-core, GPU and an oracle (implicit A). Eight threads are used for the multi-core routines, and the GPU kernel is half-warp DHTELL, the strongest of those reviewed in the previous section. The average speed-up per invocation on 8 cores is 1.96, whilst the oracle for A is only 0.53 times as fast as the standard serial version by the same measure. These results bear little relation, however, to the times required for solution, which show no clear pattern. The GPU module is unable to solve the largest of these instances, and performance is relatively poor, with invocations only 1.22 times faster on average than the serial CPU case.

The speed of solution is exceptional for all problems and modules. Gondzio [50] reported simplex solution times of 22 hours for a dimension 20 quadratic assignment problem, with a dimension 30 instance unsolved after 28 days. Similarly, although an interior point method completed the dimension 20 problem in less than half an hour, there was insufficient memory available to even attempt to solve the dimension 30 instance.

The second table shows the accuracy of the solutions returned, in terms of the maximal primal and dual infeasibilities reported at termination. As can be seen, this method is unable to find a solution to any problem that is primal feasible to within reasonable tolerances, and the particular choice of linear algebra module appears to play little to no part in this difficulty. More importantly, for the unsolved instances, the solutions returned are dual infeasible, and so cannot be used to bound the optimal objective.

4.6.3 Quantum theory subproblems

The problems in this section derive from the study of non-classicality thresholds for multiqubit states [58]. They are subproblems whose solutions drive a higher-level search for optimality, and this has not been run to termination on larger problem sizes owing to the difficulty of the generated linear instances.

Results are given for all of the subproblem instances to which access was available. There is no oracle for these problems, so only the general purpose linear algebra modules can be evaluated. Once again, the speed of termination in all cases is exceptional. Gondzio [50, see also 53] reported solution time of over 110 hours for the instance 64KX64K0 using the simplex method, and there was not enough memory available to attempt solution using an interior point method. In contrast, the longest time recorded here is around an hour.

However, as for the quadratic assignment programs, hmf is unable to locate primal feasible solutions to any of these problems, and in some cases the result is so inaccurate as to be meaningless (e.g. 16KX16K3). When time per invocation of the linear algebra operations is considered, the average speed-up is 3.37 on eight cores, and the GPU is 8.61 times faster than the serial kernels. Despite this, the tables show that these modules have very little effect on the time required for solution overall, and the solution process itself appears chaotic.

Kernel		Mean time per multiply (ms)								Geometric
Type	Threads	16KX16K0	64KX64K0	96X128-0	256X256-0	ESC16A	ESC32A	ESC64A	TAI80A	mean
CSR	1	9.13	70.51	202.80	532.61	0.36	7.85	129.23	318.15	41.09
CSR	16	1.37	8.25	22.45	63.27	0.11	1.38	21.42	53.22*	6.42
CSR	32	1.04	6.53	18.35	52.71	0.18	1.71	22.99	53.29	6.32
CSR	64	0.99	6.20	16.28	43.11*	0.28	2.26	24.20	60.34	6.69
CSR	128	1.34	7.19	16.62	46.16	0.41	3.14	28.35	67.36	8.07
CSR	256	2.11	10.70	20.77	58.52	0.83	6.58	51.67	112.52	13.11
DHELL	1	1.02	7.18	18.66	61.53	0.14	1.79	32.30	81.79	6.98
DHELL	8	0.71	5.47	14.92	46.88	0.10*	1.22*	21.47	55.00	5.01
DHELL	16	0.65	5.09	14.65	50.12	0.12	1.31	21.35*	53.90	5.05
DHELL	32	0.74	5.66	17.03	60.84	0.16	1.69	23.60	58.11	5.98
DHTELL	16	0.54*	4.38*	14.22	45.09	0.11	1.39	21.53	53.76	4.80*
DHTELL	32	0.64	4.58	15.26	50.28	0.18	1.44	21.48	53.83	5.37
DHTELL	64	0.78	4.89	13.77*	45.10	0.29	1.92	24.20	59.77	6.09

GPU kernel results on ***grunty*** *from § 4.6.1.*

Subproblem				hmf (1)		hmf (8)		hmf (GPU)		hmf (IA)	
Name	Rows	Columns	Nonzeroes	Solve	SpMV	Solve	SpMV	Solve	SpMV	Solve	SpMV
ESC16A	7,712	29,056	123,392	50.93	1.45	54.41	0.60	50.61	1.28	27.17	0.67
ESC16B	7,712	29,056	123,392	29.14	0.61	27.12	0.21	27.76	0.52	24.88	0.55
ESC16C	7,712	29,056	123,392	50.40	1.42	47.12	0.50	48.49	1.18	29.48	0.84
ESC16D	7,712	29,056	123,392	71.37	2.14	67.08	0.72	68.20	1.77	29.31	0.82
ESC32A	63,552	493,056	2,033,664	1149.47	88.20	1053.52	47.53	1045.82	43.42	896.34	110.18
ESC32B	63,552	493,056	2,033,664	403.04	27.23	390.02	14.61	407.10	13.72	336.27	28.94
ESC32C	63,552	493,056	2,033,664	1015.63	83.81	989.31	44.73	995.26	40.04	828.41	100.88
ESC32D	63,552	493,056	2,033,664	797.21	65.24	775.26	35.73	784.78	31.36	394.81	37.64
ESC64A	516,224	8,132,608	33,038,336	8911.05	1107.46	7844.22	438.65			6657.80	1294.03
TAI10A	1,820	4,150	18,200	35.58	0.84	35.19	0.55	37.79	2.10	38.43	1.57
TAI12A	3,192	8,856	38,304	91.89	2.28	90.02	1.14	93.54	3.77	100.63	4.35
TAI15A	6,330	22,275	94,950	155.89	4.76	153.51	1.95	154.14	5.23	168.33	9.19
TAI17A	9,282	37,281	157,794	262.20	8.88	250.70	3.41	262.92	9.04	268.92	16.65
TAI30A	52,260	379,350	1,567,800	1362.99	77.60	1314.93	60.71	1334.87	53.67	1394.16	157.98
TAI35A	83,370	709,275	2,917,950	1738.89	117.95	1694.39	87.46	1720.94	87.24	2066.72	278.70
TAI40A	124,880	1,218,400	4,995,200	2069.45	157.98	2010.36	110.99	2027.83	108.32	2754.08	419.71
TAI50A	245,100	3,003,750	12,255,000	4539.96	452.73	4408.47	264.31			2546.77	348.50
TAI60A	424,920	6,269,400	25,495,200	3186.70	229.55	3090.92	148.78			4983.01	791.97
TAI80A	1,011,360	19,977,600	80,908,800	16980.19	1741.96	15836.82	961.63			7290.15	1464.27

Quadratic assignment problem results on ***gruntz*** *from § 4.6.2.*

Name	Subproblem Rows	Columns	Nonzeroes	hmf (1) Primal	Dual	hmf (8) Primal	Dual	hmf (GPU) Primal	Dual	hmf (IA) Primal	Dual
ESC16A	7,712	29,056	123,392	6e-4	2e-6	7e-4	2e-6	6e-4	2e-6	1e-4	1e-7
ESC16B	7,712	29,056	123,392	2e-3	3e-7	2e-3	3e-7	1e-3	3e-7	1e-4	2e-8
ESC16C	7,712	29,056	123,392	1e-3	5e-5	2e-3	5e-5	2e-3	5e-5	1e-4	7e-7
ESC16D	7,712	29,056	123,392	2e-4	2e-5	1e-4	2e-5	1e-4	2e-5	9e-5	2e-7
ESC32A	63,552	493,056	2,033,664	4e-3	4e-7	5e-3	4e-7	9e-3	1e-5	2e-4	6e-8
ESC32B	63,552	493,056	2,033,664	6e-3	2e-8	6e-3	2e-8	5e-3	2e-8	4e-5	2e-8
ESC32C	63,552	493,056	2,033,664	3e-2	4e-6	3e-2	3e-6	4e-2	3e-6	7e-5	2e-7
ESC32D	63,552	493,056	2,033,664	6e-3	7e-5	5e-3	2e-4	4e-3	7e-5	8e-5	1e-7
ESC64A	516,224	8,132,608	33,038,336	7e-3	8e-8	6e-3	8e-8			1e-4	5e-8
TAI10A	1,820	4,150	18,200	3e-2	1e-5	3e-2	1e-5	3e-2	1e-5	5e-4	1e-6
TAI12A	3,192	8,856	38,304	4e-2	1e-4	5e-2	1e-4	5e-2	1e-4	1e-3	2e-6
TAI15A	6,330	22,275	94,950	3e-2	8e-5	2e-2	7e-5	3e-2	8e-5	1e-3	2e-6
TAI17A	9,282	37,281	157,794	4e-2	2e-4	6e-2	2e-4	6e-2	2e-4	1e-3	2e-6
TAI30A	52,260	379,350	1,567,800	7e-2	1e-2	3e-2	1e-2	3e-2	1e-2	8e-4	2e-5
TAI35A	83,370	709,275	2,917,950	3e-2	1e-2	4e-2	1e-2	4e-2	1e-2	2e-3	2e-5
TAI40A	124,880	1,218,400	4,995,200	2e-2	3e-2	3e-2	3e-2	2e-2	3e-2	2e-3	2e-5
TAI50A	245,100	3,003,750	12,255,000	1e-1	2e-2	2e-2	2e-2			5e-5	1e-5
TAI60A	424,920	6,269,400	25,495,200	9e-4	1e-2	9e-4	1e-2			5e-5	4e-6
TAI80A	1,011,360	19,977,600	80,908,800	2e-3	1e-2	2e-3	1e-2			6e-5	1e-5

Quadratic assignment problem results on ***grunty*** *from § 4.6.2.*

Subproblem			hmf (1)				hmf (8)				hmf (GPU)			
Name	Rows	Columns	Solve	SpMV	Primal	Dual	Solve	SpMV	Primal	Dual	Solve	SpMV	Primal	Dual
1kx1k0	1,025	1,025	0.7	0.03	3e-4	1e-9	0.7	0.01	3e-5	1e-9	0.7	0.06	3e-5	1e-9
2kx4k0	2,305	4,097	9.6	1.02	6e-4	3e-5	8.5	0.21	1e-3	3e-5	8.9	0.31	-4e-1	1e-3
4kx4k0	4,097	4,097	48.2	1.87	1e-4	3e-6	46.4	0.31	1e-4	3e-6	46.8	0.48	2e-5	3e-6
16kx16k0	16,385	16,385	372.5	18.79	1e-1	1e-4	362.6	7.51	2e-1	5e-5	366.9	2.70	2e-1	3e-5
16kx16k1	16,385	16,385	303.5	13.15	2e+0	3e-5	295.6	4.92	2e+0	4e-5	296.5	1.98	2e+0	5e-5
16kx16k2	16,385	16,385	291.0	9.60	1e+2	8e-4	312.2	4.01	2e+2	2e-4	309.7	1.89	2e+0	3e-5
16kx16k3	16,385	16,385	333.6	16.34	2e+0	1e-3	323.3	6.38	4e+0	7e-4	323.2	2.38	2e+0	3e-4
16kx16k4	16,385	16,385	372.5	18.79	1e-1	1e-4	362.6	6.86	2e-1	3e-4	361.6	2.69	2e-1	3e-5
35kx64k0	36,865	65,537	1055.9	185.93	2e-1	7e-4	955.7	81.14	2e-1	8e-4	893.0	16.68	2e-1	1e-3
35kx64k1	36,865	65,537	1037.1	180.21	5e-1	4e-4	953.4	78.04	5e-1	3e-4	891.1	16.35	6e-1	2e-4
64kx64k0	65,537	65,537	3987.7	535.05	2e+0	8e-4	3667.3	222.15	2e+0	6e-4	3525.6	45.38	2e+0	7e-4
64kx64k1	65,537	65,537	3080.5	389.46	5e-1	9e-4	2841.8	157.81	5e-1	7e-4	2731.9	34.02	6e-1	9e-4
64kx64k2	65,537	65,537	3736.6	499.25	3e-1	6e-4	3541.6	211.94	3e-1	7e-4	3285.9	42.66	2e-1	7e-4
96x128-0	98,305	131,073	1699.7	301.97	5e-4	1e-5	1470.3	88.48	5e-4	1e-5	1431.3	26.53	5e-4	1e-5
96x128-1	98,305	131,073	1535.1	232.25	3e-3	2e-6	1367.1	68.88	3e-3	2e-6	1336.6	22.93	3e-3	2e-6
96x128-2	98,305	131,073	1795.5	350.14	2e-3	7e-5	1560.3	103.06	5e-4	2e-4	1487.1	29.02	2e-3	7e-5
256x256-0	262,145	262,145	7911.3	2292.17	1e-3	9e-8	6026.7	324.13	1e-3	9e-8	6025.0	88.45	2e-3	9e-8
256x256-1	262,145	262,145	8420.8	2438.96	5e-4	1e-7	5861.0	342.64	5e-4	2e-7	6135.0	90.84	5e-4	2e-7
256x256-2	262,145	262,145	8971.3	2739.54	5e-4	2e-4	5912.0	384.29	5e-4	2e-4	6174.9	97.32	5e-4	2e-4

Quantum theory subproblem results on ***gruntÿ*** *from § 4.6.3.*

4.7 Conclusions

The code in `hmf` itself consists primarily of accelerated linear algebra routines, and that acceleration can be considered partially successful, in that novel techniques were developed for the performance of sparse matrix-vector products on GPUs, and significant gains were realised for the quantum theory subproblems. This chapter has also demonstrated the viability of using an oracle in a matrix-free interior point method.

However, although `hmf` terminates extremely rapidly on the tested problem classes, when compared with other techniques, the results in this chapter show that accelerated linear algebra provides little further benefit in speed, and that the solutions found in any case are usually of no value. An entire year was available for tuning, during which several researchers worked on improving the accuracy of the results returned by `hmf`, but these efforts were ultimately unsuccessful. It remains unclear whether it is possible to solve any non-trivial problem to working accuracy with this code.

Chapter 5

Conclusions

This thesis describes three distinct approaches to parallel linear programming, the parallel standard simplex method (`i6` and `i8`), the parallel revised, block-angular simplex method (`i7`), and a parallel matrix-free interior point method (`hmf`). The same conclusion results from considering any of these codes: that available memory bandwidth bounds speed-up at a low multiple of the number of threads used. Given that parallelism has costs in code complexity and overhead, the introduction of significant parallel elements into linear programming solvers of these types appears unattractive at present.

This thesis nevertheless contains some contributions believed to be novel. Without exception, the following statements should be considered qualified to indicate the possibility of pre-existing material of which the author is unaware.

In Chapter 2, a novel update for a parallel standard simplex solver was described, which enables limited maximum improvement pricing. This might be generalised to other forms of pricing which are typically considered impractical. A strategy for minimising the memory traffic during the operation of such a solver was also presented. The GPU code in this chapter is the first to demonstrate the ability to solve real, numerically difficult problems.

In Chapter 3, new techniques for working with permutation matrices, for updating a Schur complement, and for maintaining a structured basis, were described. The practicality of Kaul's method was demonstrated, with new observations on its sparsity and numerical performance, and results were provided from the first known implementation showing that it is capable of reliably solving real problems. This work also contains the first description of a means to parse standard multi-commodity flow problem files without recourse to additional information.

In Chapter 4, a new representation for sparse matrices, transposed ELLPACK, and improved kernels for sparse matrix-vector multiplication with the constraint matrices of two problem classes were described. This work also demonstrated the use of an oracle for a constraint matrix in a matrix-free interior point method.

In the appendices, a new practical dual phase one method was described which simplifies recovery from infeasibility. A preliminary description of a bound-flipping ratio test for the primal simplex method was provided, including considerations of post-solve. The appendices also include computational studies of basis updates and pricing which are more comprehensive in their range than those which have been previously published.

Appendices

Introduction

These appendices bring together work not directly related to parallelism in linear programming, which nevertheless may be of some wider interest. First, an approach to combined phase one and two in the dual simplex method is described which could not be found in the literature. Its principal attraction is the comparative ease with which a recurrence of infeasibility may be dealt.

Next, a method for making use of shadow bounds in the primal simplex method is described, which again could not be found in the literature. By treating these bounds directly, significant additional presolve reductions can be enabled, whilst the inclusion of bound-flipping in the primal simplex method may be hoped to improve its performance.

The following appendix is a study of the behaviour of pricing, in particular composite and normalised pricing, in both the primal and dual simplex methods. A direct comparison of performance, in terms of iteration counts, for the range of methods implemented could not be found in the literature.

The final appendix is a computational study of the performance of various tableau and basis representations, in terms of fill and accuracy. Although the implementations compared are naïve, it may be hoped that the relative strengths of the methods remain similar, even when more advanced strategies are applied in their implementation. A study with the same scope could not be found in previously published material.

Appendix A

Obtaining a feasible vertex

A feasible point from which to begin the simplex method may be difficult to obtain for many valid programs. The standard approach in the primal simplex method is to solve a relaxation whose optima coincide with those of the original problem. In the dual simplex method, a number of techniques have been advanced. This section describes a restriction (tightening) of the problem which shares at least one optimal point with the original.

The approaches discussed are *phase one and two methods*, which retain a constant formulation of a problem throughout the solve. The alternatives are *pure phase one* methods, in which a solution to a feasible, reduced problem, called the *phase one problem*, is used as a starting point for solves of the original problem. One advantage of simultaneous methods is that in the event of numerical difficulties, recovery from a loss of feasibility is routine.

The reformulations which underpin these methods are described in subsequent sections. The primal case is well known, and is given first. The precise form of the dual case subsequently described is believed to be novel. Both cases make use of a notional quantity M, which in some systems has been given a particular numerical value. A more stable approach is to treat M symbolically, so that an expression involving M is considered in terms of its behaviour as $M \to \infty$.

A.1 The primal big-M method

Consider a linear program of the form

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & Ax \geq b \\ & x \geq 0, b > 0, \end{aligned} \tag{A.1}$$

for which a feasible point is unknown. However, the relaxed problem, formed by introducing *artificial variables* α,

$$\begin{aligned} \text{minimise} \quad & c^\top x + Me^\top \alpha \\ \text{subject to} \quad & Ax + \alpha \geq b \\ & x, \alpha \geq 0, b > 0, \end{aligned} \tag{A.2}$$

has a feasible point $x = 0$, $\alpha = b$. The term $Me^\top\alpha$ added to the objective penalises the failure to satisfy the constraints in the original variables. Intuitively, for M sufficiently large, the simplex method will itself attain both feasibility and optimality in (A.1) merely by seeking optimality in (A.2).

Clearly (A.1) is infeasible precisely when (A.2) has no solutions with $\alpha = 0$, as the former is obtained by substituting $\alpha = 0$ into the latter.

Proposition A.1.1. *For M of sufficient size, (A.1) unbounded implies (A.2) is unbounded.*

Proof. Suppose (A.1) has a feasible solution x^* and an unbounded ray r^*, so that $x^* + \gamma r^*$ are feasible for all $\gamma \geq 0$, and $c^\top r^* < 0$. Then $(x^*, 0)$ and $(r^*, 0)$ are a solution and unbounded ray respectively for (A.2). □

Suppose that (A.2) is unbounded, with solution (x^*, α^*) and unbounded ray (r^*, ρ^*), with $c^\top r^* + Me^\top\rho^* < 0$ by assumption. Now $\rho^* \geq 0$, which implies $c^\top r^* < 0$, and feasibility of the unbounded ray depends on

$$Ax^* + \gamma Ar^* + \alpha^* + \gamma\rho^* \geq b, \tag{A.3}$$

for all $\gamma \geq 0$, so that if $Ar^* \geq 0$ then $(r^*, 0)$ is also an unbounded ray for (A.2). Otherwise, $\rho^* \geq -Ar^*$ implies $\rho^* > 0$ and so for

$$M \geq -\frac{c^\top r^*}{e^\top\rho^*} \tag{A.4}$$

this ray is no longer an improving direction. If there exists M^* such that for $M \geq M^*$ all rays of (A.2) have $\rho = 0$, then by setting $M = M^*$, a feasible point from (A.1) can be combined with any ray from (A.2) to give a proof of unboundedness for the original problem.

However, no such M^* necessarily exists. Consider the counterexample

$$\begin{aligned} \text{minimise} \quad & -x_1 \\ \text{subject to} \quad & -x_1 + x_2 \geq 1 \\ & x_1, x_2 \geq 0, \end{aligned} \tag{A.5}$$

with the relaxed formulation

$$\begin{aligned} \text{minimise} \quad & -x_1 + M\alpha_1 \\ \text{subject to} \quad & -x_1 + x_2 + \alpha_1 \geq 1 \\ & x_1, x_2, \alpha_1 \geq 0. \end{aligned} \tag{A.6}$$

A feasible solution to the relaxed problem is $(0, 1, 0)$, and $\forall\gamma$

$$(0, 1, 0) + \gamma(1, 1 - \varepsilon, \varepsilon) \tag{A.7}$$

is also feasible for all $0 \leq \varepsilon \leq 1$, and in particular for $0 < \varepsilon < 1/M$ we have

$$(-1, 0, M)^\top(1, 1 - \varepsilon, \varepsilon) = -1 + M\varepsilon < -1 + \frac{M}{M} = 0 \tag{A.8}$$

so that for any M there exists an unbounded ray for this problem which is nonzero in the artificial variables.

Note also that the unboundedness of (A.2) does not imply the feasibility of (A.1). Consider the trivial counterexample

$$\begin{aligned} \text{minimise} \quad & -x_2 \\ \text{subject to} \quad & -x_1 \geq 1 \\ & x_1, x_2 \geq 0, \end{aligned} \tag{A.9}$$

with the relaxed formulation

$$\begin{aligned} \text{minimise} \quad & -x_2 + M\alpha_1 \\ \text{subject to} \quad & -x_1 + \alpha_1 \geq 1 \\ & x_1, x_2, \alpha_1 \geq 0. \end{aligned} \tag{A.10}$$

A solution and unbounded ray for this problem are $(0, 0, 1)$ and $(0, 1, 0)$ respectively, but the original problem is infeasible.

The difficulty in interpreting unboundedness of (A.2), as highlighted in the preceding discussion, can be overcome in practice by careful selection of incoming columns, for example to reduce $\|\alpha\|$.

Proposition A.1.2. *If (A.1) is feasible, then the sets of optimal solutions of (A.1) and (A.2) are isomorphic for M of sufficient size.*

Proof. The feasible region for (A.1) is contained within that for (A.2), so that if z is the optimal objective for (A.1) and z^* is the optimal objective for (A.2), we must have $z^* \leq z$. By assumption, (A.1) has a solution, providing an upper bound z^{**} on its maximal optimal objective.

Suppose (x^*, α^*) is an optimal solution to (A.2) with $\alpha^* > 0$, then as $M \to \infty$ so $z^* \to \infty$, which contradicts the bound z^{**} previously derived. Hence, for M sufficiently large, $\alpha^* = 0$.

For any optimal solution x to (A.1), the point $(x, 0)$ is feasible for (A.2). There can be no point $(x^*, 0)$ with superior objective, as x^* would then be a point with superior objective in (A.1), contradicting optimality of x. Thus $(x, 0)$ is optimal for (A.2).

An identical argument holds for optimal solutions $(x^*, 0)$ of (A.2), and there is a trivial mapping between optimal solutions of (A.1) and (A.2) which is injective and surjective, hence an isomorphism. □

In practice, a working problem for the primal simplex method is most readily of the form

$$\begin{aligned} \text{minimise} \quad & c^\top x + Mc_I(x, \ell, u)^\top x + Mc_I(s, -U, -L)^\top s \\ \text{subject to} \quad & Ax + s = 0 \\ & x, s \text{ free} \end{aligned} \tag{A.11}$$

where c_I is a piecewise linear function, defined component by component as

$$\big\{c_I(x, \ell, u)\big\}_i = \begin{cases} -1 & x_i < \ell_i \\ 1 & x_i > u_i \\ 0 & \text{otherwise} \end{cases} \tag{A.12}$$

The artificials α are implicit in this formulation, being assumed to come into existence to retain feasibility whenever a variable moves outside its bounds. The coefficients (c, c_I) in the objective function are stored separately. Pricing with both c and c_I simultaneously is discussed in Appendix C.2.

A.2 The dual big-M method

Consider the problem

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & Ax \geq b \\ & x \geq 0, \;\; c < 0, \end{aligned} \tag{A.13}$$

for which a dual feasible point is not known. A restriction, formed by introducing upper bounds M on all variables x, can be written in normal form as

$$\begin{aligned} \text{minimise} \quad & -c^\top \beta + Mc^\top e \\ \text{subject to} \quad & -A\beta \geq b - AMe \\ & -\beta \geq -Me \\ & \beta \geq 0, \;\; c < 0. \end{aligned} \tag{A.14}$$

where $\beta = Me - x$ is a variable substitution moving x to these new bounds. The point $\beta = 0$ is a dual feasible solution for (A.14).

Observe, trivially, that (A.14) can never be unbounded for any finite M, as the greatest feasible value any variable can take is at most M.

Proposition A.2.1. *For M of sufficient size,*

i. (A.13) *is infeasible if and only if* (A.14) *infeasible.*

ii. (A.13) *is unbounded if and only if* (A.14) *has no optimal solution with* $\beta > 0$.

Proof. The feasible region of (A.13) contains that of (A.14), so that infeasibility of (A.13) trivially implies infeasibility of (A.14).

Suppose (A.13) is feasible, having the valid solution x. Let $\mu = \|x\|_\infty$ be the largest component of x, and set $M > \mu$. Now $\beta = Me - x$ satisfies

$$\begin{aligned} \beta = Me - x > \mu e - x \geq 0 \\ -\beta = x - Me \geq -Me \\ -A\beta = Ax - AMe \geq b - AMe \end{aligned} \tag{A.15}$$

so that β is a feasible solution for (A.14).

Suppose now that (A.13) is unbounded, having solution x^* and unbounded ray r^*, so that $x^* + \gamma r^*$ is feasible for all $\gamma \geq 0$ and $c^\top r^* < 0$. Observe $r^* \geq 0$ necessarily or the positivity constraints on x would prevent it from being an unbounded ray. For any solution $\beta > 0$ to (A.14),

$$-c^\top \beta = -c^\top (Me - x) > c^\top x + c^\top \gamma r^* - c^\top Me = -c^\top (\beta - \gamma r^*) \tag{A.16}$$

where $(\beta - \gamma r^*)$ is feasible for (A.14) when $\gamma \leq \min |\beta_i / r_i^*|$. Hence there can be no optimal solution to (A.14) with $\beta > 0$.

Conversely, suppose (A.13) is feasible and bounded, with optimal solution x^* and objective z^*. Note that as the feasible region of (A.14) is contained in that of (A.13), z^* is a lower bound on the objective which can be obtained in (A.14).

Let $\mu = \|x\|_\infty$ and set $M > \mu$. Then $\beta^* = Me - x^*$ is a feasible solution to (A.14) with objective z^*, and hence optimal. Now

$$\beta^* = Me - x^* > \mu e - x^* \geq 0 \tag{A.17}$$

so that $\beta^* > 0$ as required. □

Proposition A.2.2. *Every optimal solution to* (A.13) *has a corresponding optimal solution in* (A.14) *for some value of* M.

Proof. Given an optimal solution x^* to (A.13), let $\mu = \|x\|_\infty$ and set $M > \mu$. The solution $\beta^* = M - x^*$ is feasible, as shown in Proposition A.2.1, and has the same objective as x^*, hence is optimal for the restricted problem.

To see that $\|x\|_\infty$ is bounded for all optimal solutions, note that $x \geq 0$ and $c < 0$ so that

$$\mu = \|x\|_\infty \Rightarrow \exists i : c^\top x \leq c_i \mu \tag{A.18}$$

and optimality of x implies boundedness of the right-hand side. □

A working problem using this approach is most readily of the form

$$\begin{aligned} \text{minimise} \quad & c^\top (x + Mx_I) \\ \text{subject to} \quad & A(x + Mx_I) + (s + Ms_I) = 0 \\ & \ell \leq x \leq u, \\ & \ell_I \leq x_I \leq u_I, \\ & -U \leq s \leq -L, \\ & -U_I \leq s_I \leq -L_I. \end{aligned} \tag{A.19}$$

Let $\mathcal{B}$ be the set of basic variables and $\mathcal{N}$ the set of nonbasics, then ℓ_I and u_I are derived from ℓ and u by

$$\begin{aligned} \ell_{Ii} &= \begin{cases} 0 & \ell_i > -\infty \\ -1 & i \in \mathcal{N},\ \ell_i = -\infty \\ -\infty & \text{otherwise,} \end{cases} \\ u_{Ii} &= \begin{cases} 0 & u_i < \infty \\ 1 & i \in \mathcal{N},\ u_i = \infty \\ \infty & \text{otherwise,} \end{cases} \end{aligned} \tag{A.20}$$

and the definitions of L_I and U_I are analogous.

The bounds on the basic variables in (A.20) follow immediately from considering M to be allowed to take an arbitrarily large value. Clearly, any finite bound will be exceeded by Ms_I for sufficiently large M, so that the bound of 0 is forced.

In (A.14), an upper bound M was introduced on dual infeasible, nonbasic variables, which were then moved to it. It is this same bound which is represented in (A.20) by $u_I = 1$ for the phase one values x_I of the nonbasics. A variable will only move to this bound when it would otherwise have been dual infeasible, and this can be performed in identical fashion to the standard bound-flipping correction after the dual long-step ratio test.

Whenever a variable's cost ceases to be attractive, x_I may be set to zero, flipping the variable back to its finite bound. This *value correction* is the equivalent

of the cost correction which occurs in primal phase one when a basic variable which is not pivotal changes feasibility.

Note that in (A.19) primal infeasibility has a phase one and a phase two component, calculated as

$$\text{infeas}_I(x) = \begin{cases} x_I - \ell_I & x_I < \ell_I \\ x_I - u_I & x_I > u_I \\ 0 & \text{otherwise,} \end{cases} \tag{A.21}$$

$$\text{infeas}_{II}(x) = \begin{cases} x - \ell & x_I < \ell_I \ \vee \ (x_I = 0 \wedge x < \ell) \\ x - u & x_I > u_I \ \vee \ (x_I = 0 \wedge x > u) \\ 0 & \text{otherwise.} \end{cases} \tag{A.22}$$

These are trivially derived by noting that x and x_I are two parts of the same term, and hence must step to the same bound.

Appendix B

Bound flipping in the primal

In the bound-flipping ratio test for the dual simplex method [76, 43], longer steps may be performed at each iteration by observing the piecewise linear nature of the dual objective function. This in turn comes about because of the presence of boxed variables, having two finite bounds, in a practical problem formulation.

This section outlines the analogous procedure in the primal simplex method. Given the availability of dual bounds, corresponding primal bounds may be relaxed to enable longer steps to be taken. In the final part, some simple and well-known presolve operations which provide dual bounds are described, along with the means to postsolve a problem which has been treated by them. This is not a complete treatment of the subject matter, but rather an outline of the essential ideas behind it.

B.1 Shadow bounds

Suppose we are given the fully boxed primal problem

$$\begin{aligned} \text{minimise} \quad & c^\top x \\ \text{subject to} \quad & Ax \geq L \\ & -Ax \geq -U \\ & x \geq \ell \\ & -x \geq -u, \end{aligned} \tag{B.1}$$

which can be brought into normal form by the substitution $\bar{x} = x - \ell$, giving

$$\begin{aligned} \text{minimise} \quad & c^\top \bar{x} + c^\top \ell \\ \text{subject to} \quad & A\bar{x} \geq L - A\ell \\ & -A\bar{x} \geq -U + A\ell \\ & -\bar{x} \geq -(u - \ell) \\ & \bar{x} \geq 0. \end{aligned} \tag{B.2}$$

The dual of this problem is

$$\begin{aligned} \text{maximise} \quad & (L - A\ell)^\top y_1 + (A\ell - U)^\top y_2 + (\ell - u)^\top y_3 + c^\top \ell \\ \text{subject to} \quad & A^\top y_1 - A^\top y_2 - y_3 \leq c \\ & y_1, y_2, y_3 \geq 0. \end{aligned} \tag{B.3}$$

Let us suppose that a full set of *shadow bounds*, λ, Λ, ω and Ω, are available, where the reduced costs d in the primal have been bounded between Λ and Ω, and the row duals π have been bounded between λ and ω. These can be added to the dual problem, giving

$$\begin{array}{rl} \text{maximise} & L^\top y_1 - U^\top y_2 - A\ell^\top(y_1 - y_2) + (\ell - u)^\top y_3 + c^\top \ell \\ \text{subject to} & A^\top(y_1 - y_2) - y_3 \leq \quad c \\ & -A^\top(y_1 - y_2) + y_3 \leq -\Lambda \\ & (y_1 - y_2) \leq \quad \omega \\ & -(y_1 - y_2) \leq -\lambda \\ & y_3 \leq \Omega \\ & y_1, y_2, y_3 \geq 0. \end{array} \tag{B.4}$$

As a result, the primal problem becomes

$$\begin{array}{rl} \text{minimise} & c^\top x_1 - \Lambda^\top x_2 + \omega^\top x_3 - \lambda^\top x_4 + \Omega x_5 + c^\top \ell \\ \text{subject to} & A(x_1 - x_2) + (x_3 - x_4) \geq \quad L - A\ell \\ & -A(x_1 - x_2) - (x_3 - x_4) \geq -U + A\ell \\ & -(x_1 - x_2) + x_5 \qquad\quad \geq -(u - \ell) \\ & x_1, x_2, x_3, x_4, x_5 \geq 0, \end{array} \tag{B.5}$$

which resolves to

$$\begin{array}{rl} \text{minimise} & \displaystyle\sum_i f(x_i) + \sum_i g(e_i^\top A x) \\ \text{where} & f(x_i) = \left\{ \begin{array}{ll} c_i\ell_i + \Lambda_i(x_i - \ell_i) & x_i < \ell_i \\ c_i x_i & \ell_i \leq x_i \leq u_i \\ c_i u_i + \Omega_i(x_i - u_i) & x_i > u_i \end{array} \right. \\ & g(s_i) = \left\{ \begin{array}{ll} -\omega_i(s_i - L_i) & s_i < L_i \\ 0 & L_i \leq s_i \leq U_i \\ -\lambda_i(s_i - U_i) & s_i > U_i. \end{array} \right. \end{array} \tag{B.6}$$

Note that when solving with the simplex method, constraints naturally take the form $Ax + s = 0$ if both x and s have general bounds. In this case, $s = -Ax$, $-U \leq s \leq -L$, and the functions g and f are the same for both slacks and variables, i.e.

$$g'(s_i) = \left\{ \begin{array}{ll} \lambda_i\,(s_i - (-U_i)) & s_i < -U_i \\ 0 & -U_i \leq s_i \leq -L_i \\ \omega_i\,(s_i - (-L_i)) & s_i > -L_i. \end{array} \right. \tag{B.7}$$

B.2 A simplex-like method with penalties

A solver based on the primal simplex method can be written for problems of the form (B.6). The initial problem statement is taken to be

$$\begin{aligned} \text{minimise} \quad & z = \sum_{i=1}^{n} f(x_i) + \sum_{i=1}^{m} f(s_i) \\ \text{subject to} \quad & s = -Ax \\ \text{where} \quad & f(x) = \begin{cases} c_i\ell_i + \lambda_i(x_i - \ell_i) & x_i < \ell_i \\ c_i x_i & \ell_i \le x_i \le u_i \\ c_i u_i + \omega_i(x_i - u_i) & x_i > u_i \end{cases} \end{aligned} \tag{B.8}$$

and c is the original objective function, extended with zeroes to s. Here, ℓ and u are the appropriate primal bounds for each variable, and λ and ω are the appropriate shadow bounds.

It is easiest to consider each variable x_i to have two costs, an up cost γ^+ and a down cost γ^-, which change depending on the value of the variable. These can be defined as

$$\gamma_i^+ = \begin{cases} \lambda_i & x_i < \ell i \\ c_i & \ell_i \le x_i < u_i \\ \omega_i & x_i \ge u_i \end{cases} \qquad \gamma_i^- = \begin{cases} \lambda_i & x_i \le \ell i \\ c_i & \ell_i < x_i \le u_i \\ \omega_i & x_i > u_i \end{cases} \tag{B.9}$$

and give rise to corresponding up and down reduced costs δ^+ and δ^-

$$\begin{aligned} \delta_i^+ &= \gamma_i^+ - \left(\sum_{j \in P_i} \gamma_j^- \hat{a}_{ji} \right) - \left(\sum_{j \in M_i} \gamma_j^+ \hat{a}_{ji} \right) \\ \delta_i^- &= \gamma_i^+ - \left(\sum_{j \in P_i} \gamma_j^+ \hat{a}_{ji} \right) - \left(\sum_{j \in M_i} \gamma_j^- \hat{a}_{ji} \right) \end{aligned} \tag{B.10}$$

where $\hat{a}_{ij}$ is an element of the current tableau and

$$P_i = \{\, j \mid \hat{a}_{ji} > 0 \,\}, \qquad M_i = \{\, j \mid \hat{a}_{ji} < 0 \,\} \tag{B.11}$$

so that it is necessary to know the sign of the entries in the tableau corresponding to any basic variable at a bound in order to calculate an accurate reduced cost for a given column. An approximation assumes basic variables are not at bounds.

Although there are no longer any binding constraints on individual variables, the reduced cost is a convex, piecewise linear function of a variable's value. The ratio test, as for the bound-breaking phase one ratio test, performs a line search on this function.

Note that the reduced costs δ_i^+ and δ_i^-, considered as functions of x_i, change at a fixed set of points $\mathbb{B}$ for a given basis, given by

$$\mathbb{B} = \left\{ x_i + \frac{x_j - \ell_j}{\hat{a}_{ji}} \;\middle|\; j \in P_i \cup M_i \right\} \cup \left\{ x_i + \frac{x_j - u_j}{\hat{a}_{ji}} \;\middle|\; j \in P_i \cup M_i \right\} \cup \{\ell_i, u_i\}\,. \tag{B.12}$$

Thus when performing the ratio test on an incoming column, it is sufficient to find the breakpoint which makes the reduced cost zero. This breakpoint defines the leaving variable, unless it is from the incoming column, in which case a primal bound-flip occurs and there is no leaving variable.

B.3 Presolve

Brearley et al. [22] describe reductions of linear programs which may be applied before the simplex method is used, and include several techniques which might be performed were it possible to incorporate shadow bounds into the resulting problem formulation. The simplest such reduction is the removal of a singleton column

$$\begin{aligned} \text{minimise} \quad & \cdots + c_j x_j + \dots \\ \text{subject to} \quad & \cdots + a_{ij} x_j + \cdots \geq b_i, \end{aligned} \tag{B.13}$$

where $x_j \geq 0$ has a coefficient $a_{ij} > 0$ in a single constraint. In the dual problem, we have $a_{ij} y_i \leq c_j$ so that a simple upper bound can be inferred for the row dual. Now for the slack in this row, $s_i \leq -b_i$ can be replaced by an upper penalty $\omega_i = c_j / a_{ij}$. If, after solution of the problem, $s_i > -b_i$, we can set $x_j = (b_i + s_i)/a_{ij}$, restoring feasibility, otherwise $x_j = 0$ is consistent. This leaves the objective unchanged.

Suppose we infer a shadow bound Λ_j on a primal variable x_j using its constraint in the dual problem

$$\sum_i a_{ji} y_i \leq c_j \tag{B.14}$$

for which we have determined, using singleton columns, that all terms $a_{ji} y_i$ are bounded below. As a result, the primal lower bound, say 0, on x_j can be relaxed and replaced with penalties.

At the optimal solution, suppose $x_j < 0$. The penalty incurred by this violation is $x_j^\top \Lambda_j$. Each constraint i with $a_{ij} \neq 0$ contained at least one singleton column p_i, by assumption, which provided the dual bound used. If $a_{ij} < 0$ then an upper bound on y_i was used, so that there is a penalty column with $a_{ip_i} > 0$. Equivalently, $a_{ij} > 0$ implies the existence of $a_{ip_i} < 0$. Now if all of the x_{p_i} are increased by $x_j a_{ij} / a_{ip_i}$ the change in the objective is

$$\sum_i \frac{a_{ij} x_j}{a_{ip_i}} c_{p_i} = \sum_i x_j a_{ij} \frac{c_{p_i}}{a_{ip_i}} = x_j^\top \Lambda_j \tag{B.15}$$

and setting $x_j = 0$ leaves the activity of all constraints the same. Thus we have constructed a feasible solution to the original problem having the same objective.

Appendix C

Simplex pricing

The number of iterations required for the primal simplex method to solve a problem depends, to a great extent, upon the scheme used to select incoming variables. The dual simplex method is similarly dependent upon the scheme used to select leaving variables. This component of a solver is called its *pricing*.

A wide range of techniques exist which can contribute to pricing, and a practical scheme may bring together a number of them. This section presents results taken from `i4`, a standard simplex solver for which several pricing modules have been implemented. Comparisons are made of the effect of schemes for normalised pricing, and for composite pricing, on the number of iterations required to find an optimal solution.

C.1 Normalised pricing

The columns of the simplex tableau are vectors describing edges of the simplex polytope incident at the current vertex. Each nonbasic variable increases along one such edge, and its reduced cost gives the distance moved in the direction of the objective function per unit increase of the variable, which is motion along that edge.

The usual calculations in the simplex method do not lead to normalised edge vectors, so that movement along one edge may be much more rapid than movement along another. If the lengths of the edge vectors are known, normalised reduced costs can instead be calculated which represent the change in the objective for unit steps, and this in turn allows the steepest descent directions, those most closely aligned with the objective, to be selected. Techniques based on this approach are called *normalised pricing*, and may substantially reduce the number of iterations required for a solution, at the cost of making each iteration more expensive.

Steepest edge [48, 41]. In primal steepest edge pricing, the exact norms of the tableau columns are available at each iteration. In the standard simplex method, this is easily achieved. In the revised method, the weights must be updated between iterations, as the tableau is not cheaply available. Let A be the constraint matrix, and $\hat{A}$ the tableau. If x_q replaces x_p in the basis, then

the updated weights $\bar{\gamma}$ can be calculated from the existing weights γ by

$$\bar{\gamma_p} = \frac{1 + \|\hat{a}_q\|}{\hat{a}_{pq}^2} \tag{C.1a}$$

$$\bar{\gamma_i} = \gamma_i - 2\left(\frac{\hat{a}_{pi}}{\hat{a_{pq}}}\right) a_i^\top \{B^{-\top}\hat{a}_q\} + \left(\frac{\hat{a}_{pi}}{\hat{a}_{pq}}\right)^2 \|\hat{a}_q\|, \qquad i \neq p. \tag{C.1b}$$

The term $B^{-\top}\hat{a}_q$ can be found by one additional BTRAN per iteration, and the remainder of the update is similar to calculating the pivotal row.

Dual steepest edge pricing has several variants. For a standard simplex solver, it can be implemented as the transpose of the primal case. For a revised solver, exact norms of the rows of the basis inverse are usually maintained instead, and this requires only an additional FTRAN [41, 78].

Projected steepest edge [41, 56]. Suppose the space $\mathbb{F}$ is fixed to be the presently nonbasic variables, so that edge lengths in $\mathbb{F}$ are distances moved in terms of just those variables. Clearly all columns begin with a length of one, but as variables enter the basis the lengths of the columns will become the subnorms in the indices from the reference set $\mathbb{F}$. The update is much as for steepest edge, and is usually given as

$$\bar{\gamma_p} = \frac{\delta_p + \|\hat{a}_q\|_{\mathbb{F}}}{\hat{a}_{pq}^2} \tag{C.2a}$$

$$\bar{\gamma_i} = \gamma_i - 2\left(\frac{\hat{a}_{pi}}{\hat{a}_{pq}}\right) a_i^\top \left\{B^{-\top}\hat{a}_q\right\}_{\mathbb{F}} + \left(\frac{\hat{a}_{pi}}{\hat{a}_{pq}}\right)^2 \|\hat{a_q}\|_{\mathbb{F}}, \qquad i \neq p, \tag{C.2b}$$

where δ_i is one if $i \in \mathbb{F}$ and zero otherwise. Note that for exact subnorms in (C.2b), the components in $\mathbb{F}$ should be taken to be only those present both before and after the update, and an additional term of

$$\Delta = \delta_p(1-\delta_q)\hat{a}_{pi}^2 - \delta_q(1-\delta_p)\hat{a}_{qi}^2 \tag{C.3}$$

is required to correct for any change in the components in the framework.

Devex pricing [63, 56] Devex pricing is an approximate form of projected steepest edge pricing. By dropping the inner terms in (C.2b), the need to perform an additional BTRAN and row calculation in the primal, or FTRAN in the dual, is removed. The rapid accumulation of error which results can be managed by resetting the framework $\mathbb{F}$ regularly, thus returning all edge norms to 1. The update formula used is conventionally

$$\bar{\gamma_p} = \max\left(1,\ \|\hat{a}_q\|_f\right) \tag{C.4a}$$

$$\bar{\gamma_i} = \max\left(\gamma_i,\ \left|\hat{a}_{pi}^2/\hat{a}_{pq}^2\right| \|\hat{a}_q\|_f\right). \tag{C.4b}$$

Approximate steepest edge [113]. ASE is an extremely coarse approximation in which the norms are taken initially to be a count of nonzeroes per column, so that more sparse columns will be favoured. The update is a continuation of

the simplifications in Devex.

$$\bar{\gamma_p} = \frac{\gamma_q}{\hat{a}_{pq}^2} \tag{C.5a}$$

$$\bar{\gamma_i} = \max\left(\gamma_i,\ \hat{a}_{pi}^2 + 1\right) - 2\hat{a}_{pi}^2 + \frac{\gamma_q \hat{a}_{pi}}{\hat{a}_{pq}^2} \tag{C.5b}$$

C.2 Composite pricing

When both phase one and two prices are available for variables, there are essentially two approaches. Either the phase two prices can be ignored whilst phase one prices are available, or the two sets of prices can be combined throughout, for example giving primal reduced costs of $d = d_I + \mu d_{II}$. The hope is that by taking account of the true objective, the first feasible point attained by the method will be closer to optimality.

Variable mu In this simple scheme, μ begins at some fixed value (say $\frac{1}{8}$) and is reduced by one half whenever the problem appears to be optimal. The value of μ can never increase.

SOI-blended mu Here, μ again begins at some fixed value (say $\frac{1}{32}$), and a blended progress measure α, calculated by

$$\alpha_0 = 1,\ \ \alpha_i = \frac{1}{2}\left(\alpha_{i-1} + \Delta z_I\right) \tag{C.6}$$

where Δz_I is the change in the phase one objective for a given iteration, is used to control its influence. Whenever $\alpha < \rho z_I$, for some constant say $\rho = 10^{-3}$, then μ is decreased by half. Once again, μ can never increase.

ADACOMP [90, 91, 92] An attempt is made to dynamically balance the priorities of the phase one and two objectives, based on the proportion ρ of phase one improving rows or columns which are also phase two improving. When ρ is small, say $\rho < \frac{1}{3}$, so that few phase one attractive columns are also phase two attractive, μ is reduced. Conversely, when ρ is large, say $\rho > \frac{2}{3}$, so that most phase one attractive columns are phase two attractive, μ is increased.

Simplified ADACOMP [92] In this scheme, μ is chosen to enforce a fixed ratio ψ between the phase one and two objectives. The factor μ is now calculated directly at each iteration as

$$\mu = \frac{z_I}{\psi z_{II}} \tag{C.7}$$

C.3 Results

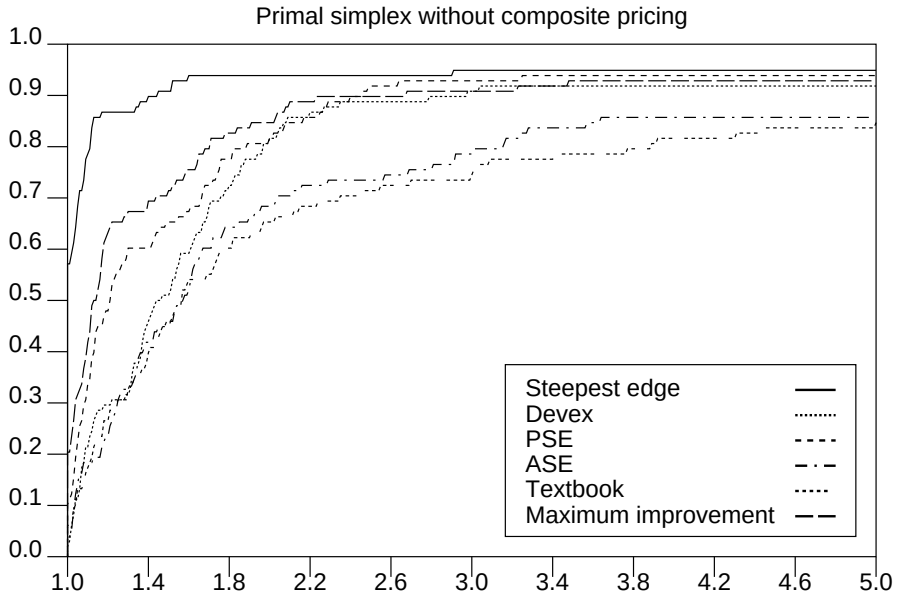
This section presents results for `i4` on the Netlib set [99], showing the effect of normalised and composite pricing on overall iteration count. As `i4` is a standard simplex code, solution times are not compared, and only a partial view of the effect of the various pricing schemes can be offered. Note that for these results, `i4` rebuilt its tableau every 200 iterations to minimise the effect of any inaccuracies which might be introduced by the standard simplex method. The dual simplex implementation of `i4` makes use of a stabilised bound-flipping ratio test with randomisation, and also the phase one method of Appendix A. A limit of 50,000 iterations was imposed, and not all solves could be completed, with numerical difficulties being another source of termination.

Results are presented as performance profiles [33]. Here, the x-axis represents the multiplier (equivalently, divisor, when minimums are sought) which separates a code's result from the best result of any code, and the y-axis represents the fraction of problems for which this code's result was within this multiple. Intercepts at the left-hand edge of such a profile are thus the proportion of problems for which a code had the best result of any code.

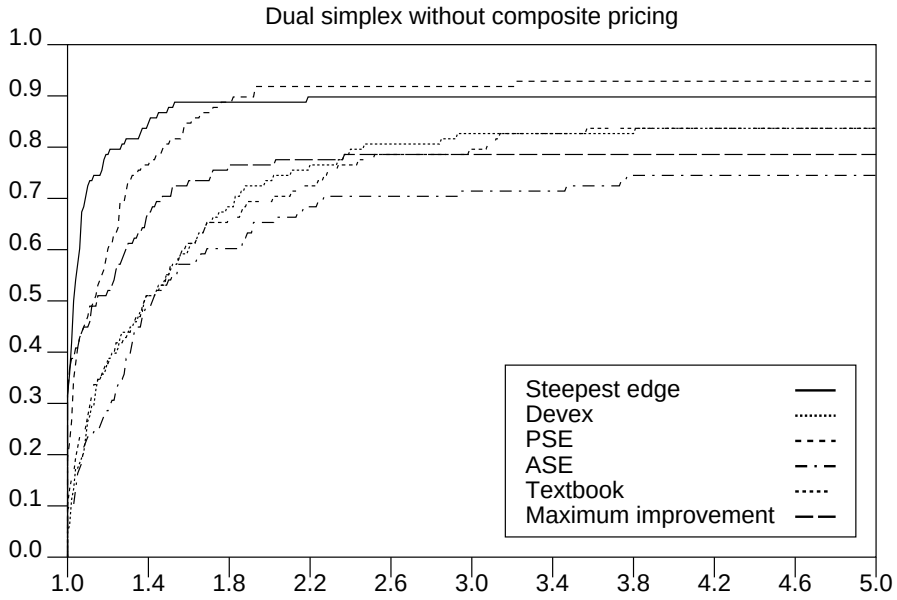
The results in Figure C.1(a) show a clear advantage for steepest edge pricing, even over maximum improvement pricing. This has significance in that the former is often presented intuitively as an approximation to the latter. Although projected steepest edge normalisation has the same cost per iteration as the exact version, it trails some distance behind it in practice. Approximate steepest edge (ASE) normalisation is by far the poorest of the pricing schemes by this measure, but has very low costs per iteration, and the effect of this is not captured here.

Figure C.1(b) shows the effect of normalised pricing for the dual simplex method. In this case, projected steepest edge pricing is much closer to the exact case, but note that now, following [41], the cost per iteration of PSE is much higher in the revised simplex method, as the projected steepest edge framework spans the entire tableau, but exact steepest edge norms correspond to rows of the basis inverse and require only an FTRAN to update. ASE was included for completeness, as no such normalised update was described in [113], with the tested version being the obvious extension of the primal case.

Composite pricing, as presented in Figure C.2, does not appear to be advantageous on this test set. For both simplex methods, the best performance is obtained without any consideration of the true objective during phase one. The scheme which is closest to being effective is SOI-blended mu for the primal simplex, which is the variant used by `i7`, but this in practice maintains μ at or near zero for the majority of problems.

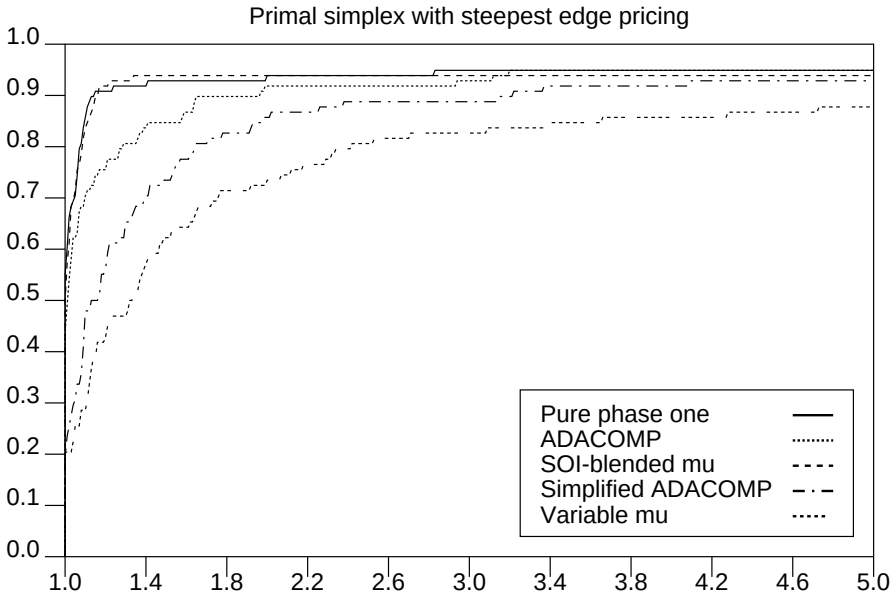


(a)

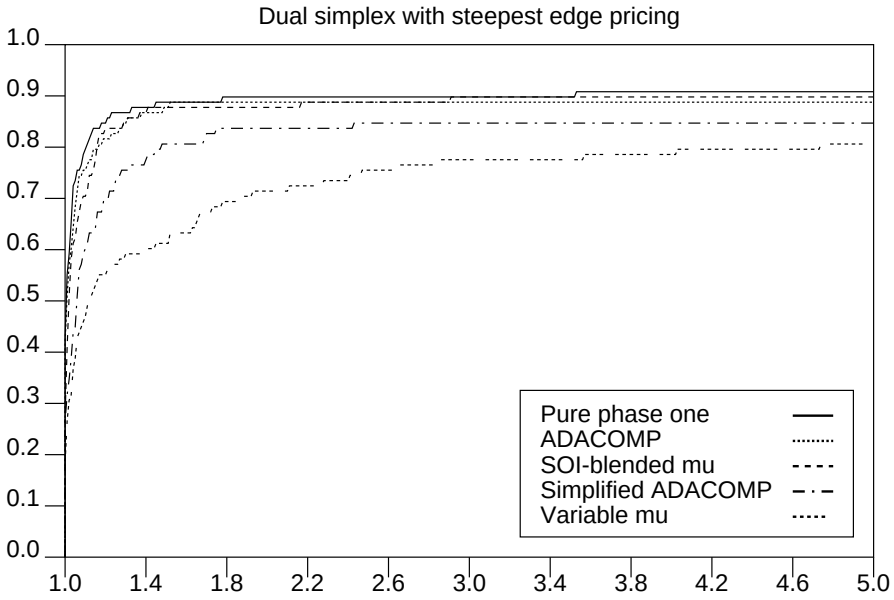


(b)

Figure C.1: Performance profiles for normalised pricing on the Netlib set. (a) Primal simplex. (b) Dual simplex.



(a)



(b)

Figure C.2: Performance profiles for composite pricing on the Netlib set. (a) Primal simplex. (b) Dual simplex.

Appendix D

The simplex tableau

The simplex method proceeds from iteration to iteration by reformulating the problem statement so that, in the absence of degeneracy, changes to the variables currently at their bounds, the *nonbasic variables*, can be evaluated in terms of changes to the variables currently within their bounds, the *basic variables*. In order for the simplex method to be performed, the coefficients of the reformulated constraints, the *tableau*, must be available.

Reformulation can be accomplished in one step by multiplying the original constraints A with the inverse of B, the matrix formed from the columns in A of the presently basic variables. In the *revised simplex method*, this procedure is used to generate parts of the tableau as they are needed, and the representation of the inverse matrix has a significant effect on the speed of such a solver.

This section presents results from `i5`, a combined standard and revised simplex solver for which a number of basis updates are available. Comparisons are made between the resulting representations in terms of their accuracy and sparsity.

D.1 The revised simplex method

Consider the problem

$$\begin{aligned} \text{minimise} \quad & [\, c \;\; 0 \,]^\top x \\ \text{subject to} \quad & [\, A \;\; I \,] x = b \\ & x \geq 0. \end{aligned} \tag{D.1}$$

At each iteration, the variables x are partitioned into a basic set $\mathcal{B}$ and a nonbasic set $\mathcal{N}$, with the columns $A_\mathcal{B}$ forming a square, invertible matrix B. Clearly, if B^{-1} is available for a given vertex, it can be used to solve the constraints for the basic variables $x_\mathcal{B}$, and the solutions substituted into the objective function, giving the complete vertex reformulation

$$\begin{aligned} \text{minimise} \quad & c_\mathcal{B}^\top B^{-1} b + (c_\mathcal{N} - c_\mathcal{B}^\top B^{-1} A_\mathcal{N}) x_\mathcal{N} \\ \text{subject to} \quad & B^{-1} A_\mathcal{N} x_\mathcal{N} + x_\mathcal{B} = B^{-1} b \\ & x \geq 0. \end{aligned} \tag{D.2}$$

The simplex method requires only the reformulated objective, right-hand side, and incoming column, and having the inverse of B is sufficient to calculate these quantities [32].

D.2 Basis representations

Numerous techniques exist for the representation of B^{-1} in the revised simplex method. This section considers a subset of them.

Dense inverse [32] An explicit inverse for B can be calculated directly at each iteration, but this would be very expensive. When reformulation using B^{-1} is applied to (D.1) an alternate view of the constraints in the resulting problem is

$$\left[(B^{-1}A) \;\; B^{-1} \right] x = b \tag{D.3}$$

so that the inverse B^{-1} is available as the part of the standard simplex tableau corresponding to the original slack variables. An explicit inverse $G = B^{-1}$ can thus be updated as a restriction of the tableau update to the slack columns. If the incoming variable is x_q which will be pivotal in row p, we have

$$g'_{ij} = \begin{cases} \dfrac{g_{ij}}{g_{pq}} & i = p \\ g_{ij} - \dfrac{g_{iq} g_{pj}}{g_{pq}} & \text{otherwise.} \end{cases} \tag{D.4}$$

The update is $O(m^2)$ operations per iteration, but multiplications must still be performed with B^{-1} to obtain the required parts of the tableau.

Elimination form inverse with product form updates [89, 30] Suppose we have an inverse of B, and we wish instead to have an inverse of B' differing in one column. Let $B' = B + (a_q - b_p)e_p^\top$ then clearly

$$B^{-1}B' = B^{-1}\left(B + (a_q - b_p)e_p^\top\right) = I + \left(B^{-1}a_q - e_p\right) e_p^\top \tag{D.5}$$

which differs in one column from the identity, and has the inverse

$$\eta = I - \frac{1}{e_p^\top B^{-1} a_q} \left(B^{-1}a_q - e_p\right) e_p^\top . \tag{D.6}$$

An inverse for B' is thus simply $B^{-1}\eta$. If this is used for the initial factorisation as well, it is called a *product form inverse* [30].

When rebuilding the inverse of B from scratch, techniques using LU factorisation are called *elimination form inverses* [89]. The observation here is that LU factors can be used to derive a sequence of product form updates which first make B upper triangular, and then reduce the upper triangular B to the identity. There are hence $2m$ elementary matrices, one per column of L and U, but they are typically more sparse than the full product form factors.

The Bartels-Golub update [10, 11] Consider the effect of a factorisation $B = LU$ on the updated basis B'. The product $L^{-1}B'$ has the form

$$L^{-1}B' = \begin{bmatrix} u_{11} & \cdots & u'_{1p} & \cdots & u_{1m} \\ & \ddots & \vdots & & \vdots \\ & & u'_{pp} & \cdots & u_{pm} \\ & & \vdots & \ddots & \vdots \\ & & u'_{mp} & & u_{mm} \end{bmatrix} . \tag{D.7}$$

To restore the triangularity of $L^{-1}B'$ the elements of u' below the diagonal, the *spike*, must be eliminated. To prevent the entire square submatrix below u'_{pp} from changing, a symmetric permutation of B is first performed to move column p to the right-hand side, and row p to the bottom, which produces a *row spike* u^*

$$L^{-1}PB'Q = \begin{bmatrix} u_{11} & \dots & u_{1p} & \dots & u_{1m} \\ & \ddots & \vdots & & \vdots \\ & & u_{pp} & \dots & u_{pm} \\ & & & \ddots & \vdots \\ & & u^*_{mp} & \dots & u^*_{mm} \end{bmatrix}. \tag{D.8}$$

The observation of Bartels and Golub [11] is that each nonzero in the row spike may be eliminated in one of two ways. If $u_{jj} \geq u^*_{mj}$ then we can subtract row j from row m to remove this nonzero, which alters the spike only in positions to the right of j, thus preserving any existing eliminations to the left. If $u_{jj} < u^*_{mj}$ then the two rows can be exchanged before the elimination is performed, so row p of U becomes the row spike. This is, in effect, a limited partial pivoting procedure, which generates elimination matrices in-between the original factors of L and U, and may cause fill in U.

The Forrest-Tomlin update [41] If partial pivoting is not performed in the LU update, then a single elimination matrix is added at each iteration to remove the row spike, and no fill occurs in U. This simplifies the update and may reduce the number of nonzeroes in the invertible representation.

The Reid update [102] If it is possible to reorder the columns of U, then symbolic pivoting can be performed prior to the Bartels-Golub update to reduce the resulting fill-in. The entering column may now be introduced at any position, and placing its last nonzero on the diagonal is attractive to reduce the length of any row spike. This gives a product $L^{-1}B'$ of the form

$$L^{-1}B = \begin{pmatrix} U_{11} & G_{12} & G_{13} \\ & G_{22} & G_{23} \\ & & U_{33} \end{pmatrix} \tag{D.9}$$

where G_{22} is the square "bump" which breaks triangularity.

Reid [102] resolves the bump in four steps. First, all column singletons are permuted to the top left corner, and absorbed into U_{11}. Second, all row singletons are permuted to the bottom right corner, and absorbed into U_{33}. Third, column singletons are once again permuted to the top left corner. At this point, the bump has been removed entirely if such was possible. The fourth step is simply the application of Bartels-Golub to whatever bump remains.

The Suhl-Suhl update [111] If the columns of U are permuted to place the last spike nonzero on the diagonal, then there is still a reduction in the lengths of the row spikes, even without the other parts of Reid's method. If both symbolic and partial pivoting is ignored, then the resulting method is comparable to the Forrest-Tomlin update, with shorter spikes traded off against fill in U.

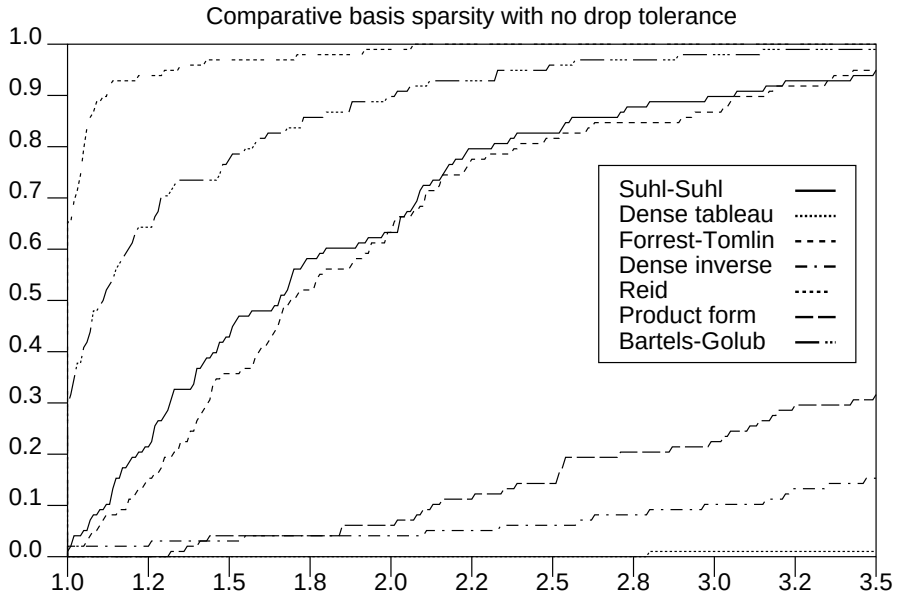
D.3 Results

This section presents results from `i5` on two problem sets, Netlib [99], and square, sparse, randomly generated problems. All revised updates ran for a fixed interval of 500 iterations between inverts, to give time for numerical error and fill to accumulate in a controlled way. A time-out of one hour was applied to each run, and not all problems were solved, with numerical difficulty being another source of non-completion. Nevertheless, this study provides an indication of the probable relative performance of the tested representations on these problems.

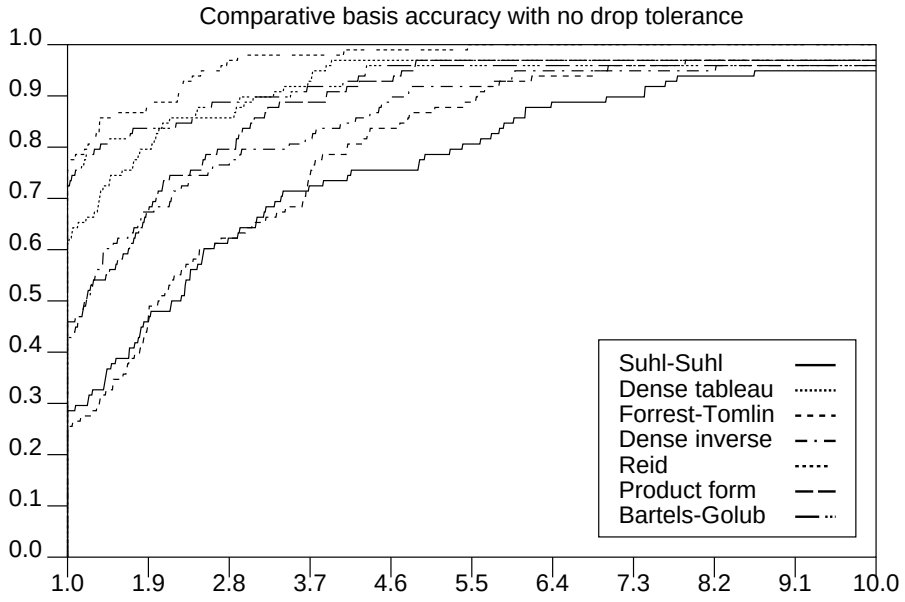
Results are again presented as performance profiles [33] (see Appendix C), this time for both basis sparsity and basis accuracy. The former is calculated as the total number of nonzeroes in the active representation, and the latter as the maximal deviation from the identity that occurred at any iteration in the product $B^{-\top}B^{\top}$, formed by BTRAN where appropriate. Numbers for accuracy are transformed as $\log_{10}(x)$, so that a multiple of two represents accuracy within a factor of 100 of the leading code.

There are 98 Netlib problems, drawn from a variety of applications, and though some are numerically challenging, few are notable for their sparsity, at least when compared to other sets of real problems. The results in Figure D.1 show that, on this set, a Reid inverse is by far the most sparse, and both a dense inverse and product form inverse appear uncompetitive in terms of nonzeroes. Forrest-Tomlin and Suhl-Suhl updates have the lowest accuracy, whereas the Reid update again has the highest.

There were 60 random problems, 30 of dimension 1000×1000 and 30 of dimension $3000x3000$, which were further divided into 1%, 5% and 10% dense instances. The performance profiles for these problems, shown in Figure D.2, bear little resemblance to those for real problems, which highlights the ongoing problem of academic codes being tested only on such formulations [15, 110, 55]. Here, all revised updates experience severe fill very early, and a dense inverse appears to be much more competitive. Accuracy is high throughout, despite the aggressive nature of the test.

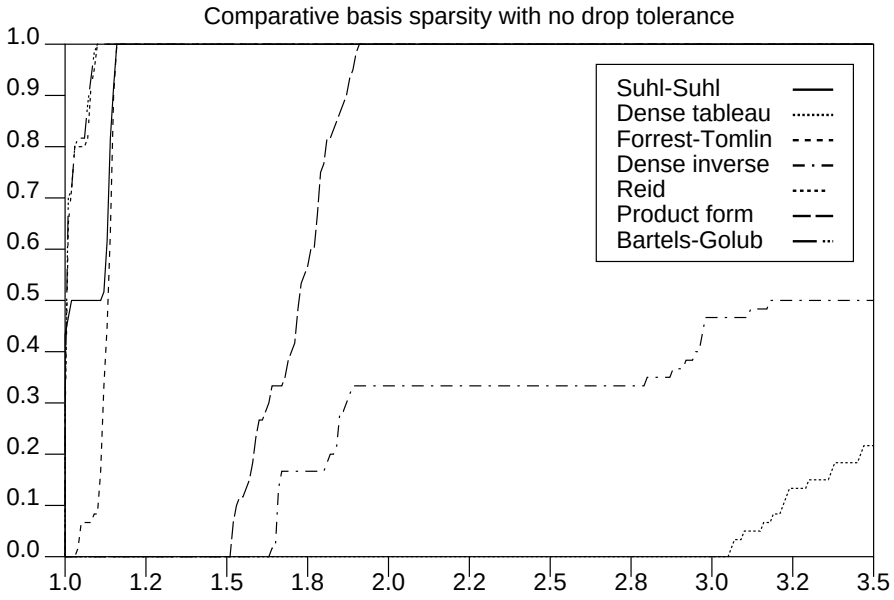


(a)

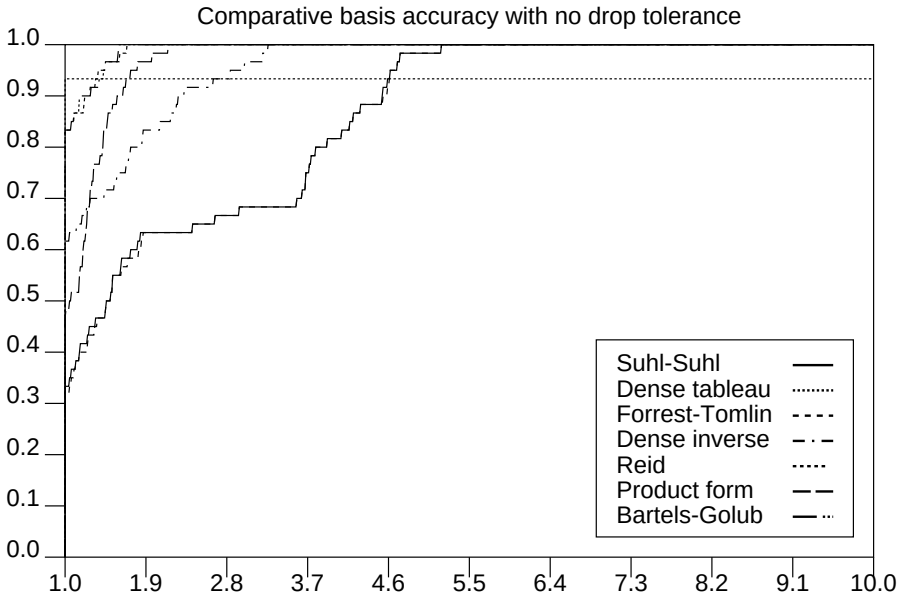


(b)

Figure D.1: Performance profiles for the problems of the Netlib set.
(a) Basis sparsity. (b) Log basis BTRAN accuracy.



(a)



(b)

Figure D.2: Performance profiles for randomly generated, sparse problems. (a) Basis sparsity. (b) Log basis BTRAN accuracy.

Appendix E

i7 - Extended results

This appendix presents results for `i7` on a standard test set for numerical stability, on multi-commodity flow problems, and on a selection of more challenging performance benchmarks [24, 95]. In the results which follow, the score α is once again a measure of accuracy (see § 2.4), calculated from the result z provided by `i7` as

$$\alpha = \left\lfloor -\log_{10} \left| \frac{z - z^*}{z^*} \right| \right\rfloor, \tag{E.1}$$

for z^* the true optimal objective. The score σ is found as

$$\sigma = \left\lfloor 100 \times \frac{t_s}{t_p} \right\rfloor \tag{E.2}$$

where t_p is the time for a parallel solve, and t_s the corresponding serial time. Note that where σ is given, it is calculated between runs with identical numbers of blocks, but only the time for the parallel run is given. It is not a comparison between two numbers in the same table.

`i7` is run with its default options, which include scaling and, where appropriate, Aykanat structurisation using hypergraph partitioning. In the following tables, a heading "`i7` (n)" means `i7` run with n blocks, or if the number of blocks is given then n superblocks, and n workers. The version of FICO Xpress Optimizer is 7.3.1. On `grunty`, the version of `clp` is 1.06.00 and `glpk` is 4.29. On `richtmyer`, the version of `clp` is 1.12.0 and the version of `glpk` is 4.43. For all solvers, presolve and crash are disabled, as creating competitive versions for `i7` was beyond the scope of this work.

E.1 Standard test problems

The Netlib set [99] is a standard collection of problems for linear programming codes, and provides a useful test of numerical stability and resilience. The following results are important to demonstrate the effectiveness of the previously described techniques in a practical revised simplex solver.

`i7` successfully solved 98% of the Netlib set to $\alpha \geq 5$, with the optimization of the remaining two problems, `QAP12` and `QAP15`, being incomplete after the maximum allowed time of 30 minutes. `glpk` is also unable to solve `QAP15` within this time. For those runs which were completed, the time needed for `i7` to solve

each problem is broadly comparable with that taken by the other solvers. That i7 is the only code making use of a product form update, and that this is more prone to fill than other basis updates (see Appendix D), may go some way to explaining the difficulties encountered for the QAP instances, which are known to be linear relaxations of quadratic assignment problems (see § 4.4).

When both the solution times and the achieved α scores are considered together, these results demonstrate that the decomposed basis representation in i7, when compared to i7's own traditional factored inverse, does not lead to reduced accuracy. If there were significant compromises in numerical stability entailed by the structured form, some performance degradation or loss of significant figures in the solution would be expected.

The like-for-like speed-up on this set, that is the increase in performance from applying more workers to the same structured form, peaks at 1.45 on sixteen cores for STOCFOR3, which is a speed-up of 3.07 times over the unstructured solve. The mean like-for-like speed-up is just 0.48 on sixteen cores, however, and just 0.41 times over the unstructured case. These losses can be attributed to the small size of the problems in the Netlib set.

E.2 Multicommodity flow problems

The problems in this section are taken from a repository of multi-commodity flow problems in JLF format [97]. In the following tables, results for the two smallest instances in this set, CHEN0 and PSP1, have been removed to save space. The remaining problems are of, at most, moderate size, with the largest constraint matrix being of dimension $97,753 \times 259,670$, for ALK.TWO, and the largest number of blocks being 194, for JL209.

i7 follows the same solution path for any number of workers if the block-angular formulation is constant, and as the natural blocks in these problems are available, only the time taken for different numbers of workers to complete each solve is given. The other optimizers are provided with linear programming formulations identical to those used by i7. Note that i7 and the tool graph from [97] produce formulations leading to different optimal objective values for four of these problems, JL023, JL147, ALK.HALF and ALK.TWO. The first two result from a rounding error in graph, but the cause of the discrepancies for the ALK problems is unknown.

These instances are numerically easy, and the majority are very small. Peak speed-up is achieved for ALK.TWO at 1.23 on two cores, 1.74 on eight cores and 1.75 on sixteen cores, despite this problem having a non-trivial linking part of 4,212 rows. The mean speed-up, however, is just 0.62 on two cores, 0.61 on eight cores, and 0.52 on sixteen cores.

E.3 Larger test problems

The problems in this section are derived from three sources. Firstly, the Kennington set [24] are a collection of larger instances in the Netlib repository originating from a study of military airlift applications. They are notable for their sparsity, and may be considered numerically easy. Four of the instances tested, DCP1, DCP2, DETEQ8 and DETEQ27, are not widely available, and provide some

estimate of the performance of `i7` relative to the others on problems for which those solvers cannot have been tuned. The remaining instances are drawn from a public benchmark set [95], and were selected to be solvable by `i7` in under 30 minutes.

`i7` is broadly competitive with the non-commercial solvers `clp` and `glpk` on this set. Like-for-like speed-up on `richtmyer` peaks at 1.93 for `KEN-18` with sixteen cores, and 2.06 for `PDS-40` with eight cores, which represent speed-ups of 3.97 and 0.92 respectively over the unstructured case. Mean like-for-like speed-up is 1.04 on sixteen cores, and 1.09 on eight cores, with mean speed-up over the unstructured case of 1.23 and 1.28 respectively. On `grunty`, like-for-like speed-up peaks at 2.11 for both `PDS-40` and `KEN-18`, and these represent a speed-up of 1.13 and 3.08 respectively over the unstructured cases. Mean like-for-like speed up is 1.24 on eight cores, and 1.39 over the unstructured case.

The problems in this test set are substantially larger and more difficult than those discussed previously. The structured form in fact appears more efficient than `i7`'s traditional basis representation for these instances, though the possibility that this is due to chance, with solves in the different conditions following different paths, cannot be excluded. Speed-up is poor for many problems, but this can be attributed to the difficulty of finding an efficient block-angular form, there being no particular reason for these general instances why such a form should exist. For example, `CRE-D` is one problem showing a a slow-down on sixteen cores, but the best block-angular form found has 4,043 linking rows, when the entire problem has just 8,949 rows, and many of the blocks in the resulting formulation are trivial. The overhead of parallelism for such problems appears to be significant.

	Problem		i7		i7 (2)		i7 (8)		i7 (16)		glpk	clp	Xpress
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)	Time (s)
25FV47	821	1571	0.59	11	0.64	11	0.55	11	0.61	11	0.3	0.27	0.25
80BAU3B	2262	9799	0.83	11	0.83	11	0.65	11	0.66	11	1.0	0.34	0.40
ADLITTLE	56	97	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
AFIRO	27	32	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
AGG	488	163	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
AGG2	516	302	0.00	11	0.01	11	0.00	11	0.01	11	0.0	0.00	0.00
AGG3	516	302	0.00	11	0.01	11	0.01	11	0.01	11	0.0	0.00	0.00
BANDM	305	472	0.02	12	0.02	12	0.02	12	0.03	12	0.0	0.02	0.02
BEACONFD	173	262	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
BLEND	74	83	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
BNL1	643	1175	0.13	10	0.12	10	0.14	10	0.16	10	0.1	0.08	0.16
BNL2	2324	3489	1.12	10	0.91	10	0.77	10	0.77	10	0.6	0.48	0.63
BOEING1	351	384	0.02	11	0.01	11	0.02	11	0.02	11	0.0	0.01	0.02
BOEING2	166	143	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
BORE3D	233	315	0.00	11	0.00	11	0.01	11	0.01	11	0.0	0.00	0.00
BRANDY	220	249	0.01	11	0.01	11	0.01	11	0.01	11	0.0	0.01	0.01
CAPRI	271	353	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.01	0.01
CYCLE	1903	2857	0.05	11	0.12	11	0.07	11	0.16	11	0.1	0.18	0.03
CZPROB	929	3523	0.08	10	0.10	10	0.10	10	0.09	10	0.1	0.05	0.06
D2Q06C	2171	5167	5.27	6	5.80	6	3.80	6	4.01	6	2.9	2.35	2.46
D6CUBE	415	6184	6.78	10	6.28	10	9.64	10	8.29	10	1.7	1.89	2.74
DEGEN2	444	534	0.10	14	0.09	15	0.08	15	0.11	16	0.1	0.04	0.05
DEGEN3	1503	1818	1.91	13	1.21	15	1.34	15	1.23	14	0.7	0.63	0.67
DFL001	6071	12230	379.20	11	332.88	10	278.06	10	201.94	10	61.6	22.19	1038.99
E226	223	282	0.01	9	0.01	9	0.01	9	0.01	9	0.0	0.01	0.01
ETAMACRO	400	688	0.02	8	0.02	8	0.02	8	0.02	8	0.0	0.02	0.01
FFFFF800	524	854	0.03	11	0.03	11	0.03	11	0.04	11	0.0	0.03	0.02
FINNIS	497	614	0.01	6	0.01	6	0.01	6	0.01	6	0.0	0.02	0.01
FIT1D	24	1026	0.04	11	0.04	11	0.04	11	0.04	11	0.0	0.04	0.03
FIT1P	627	1677	0.08	11	0.11	11	0.12	11	0.11	11	0.1	0.11	0.08
FIT2D	25	10500	4.21	11	4.22	11	4.20	11	4.22	11	4.4	1.73	0.52
FIT2P	3000	13525	6.11	9	21.18	9	22.27	9	18.47	9	7.5	1.30	1.29
FORPLAN	161	421	0.01	11	0.01	11	0.01	11	0.01	11	0.0	0.01	0.01

Netlib results on `richtmyer` *from § E.1. Continued on the next page.*

Problem			i7		i7 (2)		i7 (8)		i7 (16)		glpk	clp	Xpress
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)	Time (s)
GANGES	1309	1681	0.06	11	0.07	11	0.07	11	0.07	11	0.1	0.05	0.02
GFRD-PNC	616	1092	0.02	11	0.03	11	0.03	11	0.03	11	0.0	0.02	0.01
GREENBEA	2392	5405	1.88	11	2.19	11	1.60	11	1.50	11	1.8	2.32	1.81
GREENBEB	2392	5405	1.67	11	1.44	11	1.19	11	1.02	11	1.5	1.04	0.83
GROW7	140	301	0.02	11	0.02	11	0.02	11	0.01	11	0.0	0.00	0.01
GROW15	300	645	0.09	10	0.18	10	0.49	10	0.53	10	0.1	0.02	0.10
GROW22	440	946	2.05	10	0.38	10	1.07	10	0.41	10	0.2	0.04	0.12
ISRAEL	174	142	0.01	11	0.01	11	0.01	11	0.01	11	0.0	0.00	0.01
KB2	43	41	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
LOTFI	153	308	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
MAROS	846	1443	0.11	11	0.12	11	0.11	11	0.11	11	0.1	0.09	0.09
MAROS-R7	3136	9408	2.08	10	2.89	10	3.03	10	4.08	10	2.5	2.04	1.61
MODSZK1	687	1620	0.12	10	0.11	10	0.10	10	0.09	10	0.1	0.05	0.05
NESM	662	2923	0.28	10	0.25	10	0.30	10	0.29	10	0.4	0.17	0.12
PEROLD	625	1376	0.47	11	0.40	11	0.38	11	0.43	11	0.2	0.16	0.43
PILOT	1441	3652	7.70	5	8.01	5	5.85	5	6.65	5	2.7	3.13	2.31
PILOT.JA	940	1988	1.82	6	1.30	6	1.32	8	1.33	6	0.3	0.38	1.60
PILOT.WE	722	2789	0.69	10	0.68	10	0.47	10	0.59	10	0.2	0.19	0.54
PILOT4	410	1000	0.09	11	0.13	11	0.12	11	0.12	11	0.1	0.05	0.10
PILOT87	2030	4883	18.62	5	20.48	6	15.39	6	16.48	7	6.7	8.30	5.55
PILOTNOV	975	2172	0.79	11	0.53	11	0.42	11	0.60	11	0.2	0.19	0.46
QAP8	912	1632	6.79	12	4.48	12	5.00	11	4.78	12	0.8	0.77	0.82
QAP12	3192	8856									83.3	65.24	53.54
QAP15	6330	22275										873.67	623.85
RECIPE	91	180	0.00	16	0.00	16	0.00	16	0.00	16	0.0	0.00	0.00
SC105	105	103	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
SC205	205	203	0.01	11	0.01	11	0.01	11	0.01	11	0.0	0.00	0.00
SC50A	50	48	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
SC50B	50	48	0.00	15	0.00	15	0.00	15	0.00	15	0.0	0.00	0.00
SCAGR7	129	140	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
SCAGR25	471	500	0.02	10	0.02	10	0.02	10	0.02	10	0.0	0.02	0.02
SCFXM1	330	457	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.01	0.01
SCFXM2	660	914	0.04	13	0.03	13	0.03	13	0.04	13	0.0	0.03	0.02

Netlib results on ***richtmyer*** *from § E.1.* *Continued on the next page.*

Problem		i7		i7 (2)		i7 (8)		i7 (16)		glpk	clp	Xpress
Name	Rows	Columns	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	α	Time (s)	Time (s)	Time (s)
SCFXM3	990	1371	0.07	11	0.08	11	0.06	11	0.07	11	0.1	0.05	0.04
SCORPION	388	358	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.01	0.00
SCRS8	490	1169	0.04	12	0.04	12	0.03	12	0.04	12	0.0	0.02	0.02
SCSD1	77	760	0.00	11	0.01	11	0.01	11	0.01	11	0.0	0.00	0.00
SCSD6	147	1350	0.02	10	0.03	10	0.03	10	0.03	10	0.0	0.01	0.01
SCSD8	397	2750	0.20	11	0.38	11	0.51	11	0.16	11	0.1	0.06	0.05
SCTAP1	300	480	0.00	16	0.01	15	0.01	15	0.01	15	0.0	0.01	0.00
SCTAP2	1090	1880	0.02	10	0.02	10	0.02	10	0.03	10	0.1	0.03	0.01
SCTAP3	1480	2480	0.04	16	0.05	16	0.06	16	0.05	16	0.1	0.08	0.01
SEBA	515	1028	0.03	16	0.03	16	0.04	16	0.04	16	0.0	0.01	0.01
SHARE1B	117	225	0.00	11	0.01	11	0.01	11	0.01	11	0.0	0.00	0.01
SHARE2B	96	79	0.00	11	0.00	11	0.00	11	0.00	11	0.0	0.00	0.00
SHELL	536	1775	0.01	16	0.01	16	0.02	16	0.02	16	0.0	0.01	0.01
SHIP04L	402	2118	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.01	0.01
SHIP04S	402	1458	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.00	0.00
SHIP08L	778	4283	0.03	11	0.03	11	0.03	11	0.03	11	0.0	0.02	0.01
SHIP08S	778	2387	0.01	10	0.01	10	0.02	10	0.02	10	0.0	0.01	0.01
SHIP12L	1151	5427	0.04	10	0.04	10	0.04	10	0.04	10	0.1	0.02	0.02
SHIP12S	1151	2763	0.02	11	0.02	11	0.03	11	0.03	11	0.1	0.02	0.01
SIERRA	1227	2036	0.02	10	0.03	10	0.03	10	0.03	10	0.0	0.02	0.01
STAIR	356	467	0.05	10	0.06	10	0.05	10	0.06	10	0.0	0.04	0.04
STANDATA	359	1075	0.00	16	0.00	16	0.00	16	0.00	16	0.0	0.00	0.00
STANDGUB	361	1184	0.00	16	0.00	16	0.00	16	0.00	16	0.0	0.00	0.00
STANDMPS	467	1075	0.00	15	0.01	15	0.01	15	0.01	15	0.0	0.01	0.00
STOCFOR1	117	111	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
STOCFOR2	2157	2031	0.30	11	0.22	11	0.17	11	0.23	11	0.3	0.16	0.07
STOCFOR3	16675	15695	13.54	11	11.90	11	7.44	11	6.41	11	17.1	8.20	11.27
TRUSS	1000	8806	4.67	11	4.46	11	2.81	11	3.53	11	1.4	1.32	0.62
TUFF	333	587	0.01	10	0.01	10	0.01	10	0.01	10	0.0	0.01	0.01
VTP.BASE	198	203	0.00	10	0.00	10	0.00	10	0.00	10	0.0	0.00	0.00
WOOD1P	244	2594	0.07	10	0.06	10	0.05	10	0.05	10	0.1	0.08	0.07
WOODW	1098	8405	0.21	10	0.20	10	0.20	10	0.21	10	0.3	0.26	0.12

Netlib results on `richtmyer` *from § E.1.*

Problem				i7 (1)	i7 (2)	i7 (8)	i7 (16)	glpk	clp	Xpress
Name	Rows	Columns	Blocks	Bundles	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
10TERM.0	1953	3324	10	143	0.08	0.12	0.12	0.18	0.2	0.06	0.02
10TERM.100	1937	3164	10	127	0.10	0.15	0.14	0.19	0.2	0.10	0.03
10TERM.50	1944	3234	10	134	0.09	0.13	0.13	0.18	0.2	0.05	0.03
10TERM	1956	3354	10	146	0.09	0.15	0.14	0.22	0.2	0.03	0.03
15TERM	4318	7908	15	253	0.43	0.50	0.51	0.54	1.1	0.24	0.11
15TERM.0	4267	7143	15	202	0.39	0.50	0.45	0.49	0.9	0.30	0.11
ALK.HALF	27034	73324	23	1251	3.49	3.59	2.77	2.96	29.7	5.30	0.77
ALK.TWO	97753	259670	23	4212	124.23	100.48	71.31	70.74	712.4	113.56	48.49
ASSAD1.5K	239	294	3	98	0.00	0.01	0.01	0.02	0.0	0.00	0.00
ASSAD1.6K	239	294	3	98	0.00	0.01	0.01	0.02	0.0	0.00	0.00
ASSAD3.4K	714	1224	6	204	0.06	0.09	0.10	0.14	0.0	0.02	0.04
ASSAD3.7K	714	1224	6	204	0.07	0.10	0.11	0.13	0.0	0.02	0.02
CHEN1	245	870	5	65	0.01	0.02	0.03	0.04	0.0	0.01	0.01
CHEN2	442	2506	7	155	0.05	0.09	0.10	0.12	0.1	0.03	0.01
CHEN3	521	2235	15	56	0.02	0.05	0.05	0.07	0.0	0.02	0.01
CHEN4	1001	6300	15	176	0.21	0.35	0.40	0.40	0.3	0.12	0.22
CHEN5	892	5690	10	242	0.26	0.39	0.40	0.43	0.3	0.22	0.23
CHEN6	546	3681	9	177	0.08	0.14	0.16	0.20	0.1	0.06	0.03
JL023	485	1278	18	71	0.01	0.03	0.03	0.04	0.0	0.01	0.01
JL049	2097	5480	40	137	0.09	0.14	0.14	0.16	0.2	0.09	0.05
JL141	19061	59268	132	449	2.11	2.35	1.78	1.76	15.9	1.66	0.94
JL147	21100	72800	140	520	5.04	5.61	4.51	3.93	52.3	2.52	1.91
JL158	22281	65826	138	477	2.48	2.81	1.94	1.90	19.8	5.81	1.11
JL188	31881	111718	166	673	2.69	2.74	1.70	1.91	37.2	2.76	0.63
JL207	39849	137214	189	726	44.73	49.38	36.25	31.34	235.7	10.38	21.17
JL209	41311	148410	194	765	42.70	45.65	33.85	31.93	164.0	22.07	17.64
PSP2	193	720	4	73	0.01	0.03	0.04	0.06	0.0	0.01	0.01
PSP3	243	896	8	43	0.01	0.03	0.03	0.04	0.0	0.01	0.01
PSP4	300	860	10	30	0.01	0.03	0.03	0.05	0.0	0.01	0.01
PSP5	339	1503	9	69	0.04	0.08	0.09	0.12	0.0	0.03	0.01
PSP6	609	3328	4	369	0.40	0.63	0.54	0.62	0.2	0.26	0.13
PSP7	1154	8610	6	650	3.96	5.63	5.47	5.00	1.9	1.45	1.82
VER8	9511	18830	3	301	0.77	0.94	1.02	1.39	3.9	0.52	0.14

Multicommodity flow problem results on **richtmyer** *from § E.2.*

Problem			i7	i7 (2)		i7 (8)		i7 (16)		glpk	clp	Xpress
Name	Rows	Columns	Time (s)	Time (s)	σ	Time (s)	σ	Time (s)	σ	Time (s)	Time (s)	Time (s)
CRE-A	3516	4067	0.52	0.62	79	0.52	79	0.76	64	0.7	0.24	0.35
CRE-B	9648	72447	17.64	18.34	81	15.47	81	12.52	82	52.6	20.17	4.05
CRE-C	3068	3678	0.53	0.61	73	0.50	73	0.68	73	0.5	0.26	0.29
CRE-D	8926	69980	17.09	17.86	76	15.00	76	19.20	70	47.7	24.22	4.35
KEN-07	2426	3602	0.14	0.27	63	0.28	63	0.28	60	0.3	0.13	0.04
KEN-11	14694	21349	5.12	5.15	94	3.81	94	3.95	106	14.4	3.56	1.02
KEN-13	28632	42659	41.61	29.78	101	14.97	101	14.33	135	80.1	17.58	6.28
KEN-18	105127	154699	634.95	531.27	114	257.70	114	159.82	193	1358.1	211.13	75.22
OSA-07	1118	23949	0.20	0.16	70	0.23	70	0.18	56	0.2	0.07	0.08
OSA-14	2337	52460	0.86	0.76	74	0.67	74	0.66	71	0.9	0.20	0.55
OSA-30	4350	100024	4.41	3.35	83	2.36	83	2.35	91	3.2	0.56	1.76
OSA-60	10280	232966	14.44	15.42	86	11.71	86	11.37	88	16.9	2.10	4.50
PDS-02	2953	7535	0.10	0.17	64	0.17	64	0.16	66	0.3	0.08	0.03
PDS-06	9881	28655	1.49	1.54	89	1.13	89	1.48	92	4.9	0.77	0.72
PDS-10	16558	48763	4.73	5.88	96	5.20	96	5.04	101	17.3	3.48	1.73
PDS-20	33874	105728	40.52	49.50	110	62.00	110	42.96	144	197.9	28.83	43.33
DANO3MIP_LP	3202	13873	116.53	199.39	96	140.67	101	142.47	101	10.8	14.01	11.12
GEN4	1537	4297	159.59	99.81	110	104.16	100	105.86	100	50.4	44.12	37.17
LP22	2958	13434	742.89	315.97	107	225.81	119	1099.71	109	54.0	32.58	26.82
PDS-40	66844	212859	283.09	477.55	118	305.56	206	244.72	175	2073.3	161.41	191.77
RLFPRIM	58866	8052	79.54	12.54	104	17.98	123	16.08	117	8.9	4.77	2.28
STORMG2-27	14441	34114	2.38	4.02	93	3.00	119	3.53	106	17.1	4.28	0.88
STORMG2-125	66185	157496	55.80	87.05	104	52.23	153	43.08	168	490.8	121.42	18.76
NUG08	912	1632	5.86	5.48	94	5.62	93	6.58	93	0.8	0.74	1.66
NSCT2	23003	14981	3.61	7.21	123	6.74	129	6.97	119	9.2	1.75	3.62
SGPF5Y6	246077	308634	287.51	135.07	104	119.60	179	97.49	180	1820.6	82.32	8.08
WORLD	35510	32734	612.92	431.98	129	217.05	150	243.57	145	387.3	178.11	370.69
DCP1	4950	3007	1.14	1.10	98	0.76	100	0.96	94		1.61	1.18
DCP2	32388	21087	81.35	96.92	154	39.00	169	37.30	154		70.03	57.02
DETEQ8	20678	56227	16.68	21.73	99	15.82	121	17.73	112	34.4	9.27	0.96
DETEQ27	68672	186928	174.58	202.62	109	97.21	151	80.48	164	397.5	100.04	6.46

Larger test problem results on `richtmyer` *from § E.3.*

Problem			i7	i7 (2)		i7 (4)		i7 (8)		glpk	clp	Xpress
Name	Rows	Columns	Time (s)	Time (s)	σ	Time (s)	σ	Time (s)	σ	Time (s)	Time (s)	Time (s)
CRE-A	3516	4067	0.66	0.94	74	0.75	91	0.81	92	1.3	0.46	0.47
CRE-B	9648	72447	30.13	24.57	109	23.55	115	22.63	122	40.6	143.64	6.37
CRE-C	3068	3678	0.69	0.78	81	0.72	89	0.84	86	1.2	0.51	0.39
CRE-D	8926	69980	29.86	27.13	96	18.34	117	18.24	123	43.2	185.38	6.64
KEN-07	2426	3602	0.20	0.30	79	0.33	73	0.43	70	1.1	0.18	0.05
KEN-11	14694	21349	8.02	7.52	102	5.14	129	5.08	149	54.7	6.10	1.52
KEN-13	28632	42659	61.83	43.09	105	27.49	139	23.32	151	375.9	26.83	9.69
KEN-18	105127	154699	1051.25	899.53	114	468.37	164	341.06	211	6229.5	346.05	136.67
OSA-07	1118	23949	0.29	0.26	67	0.21	71	0.34	51	0.6	0.10	0.14
OSA-14	2337	52460	1.45	1.26	71	0.94	89	0.98	89	2.7	0.33	1.08
OSA-30	4350	100024	8.68	5.54	84	4.70	106	3.91	106	10.6	0.91	4.97
OSA-60	10280	232966	27.09	26.35	88	19.02	112	18.39	121	54.2	4.02	10.27
PDS-02	2953	7535	0.15	0.28	54	0.27	61	0.27	65	0.8	0.15	0.04
PDS-06	9881	28655	2.37	2.65	82	2.40	108	1.67	117	9.1	2.06	1.15
PDS-10	16558	48763	7.37	10.28	87	7.06	120	7.05	135	46.4	3.67	2.92
PDS-20	33874	105728	70.09	84.98	108	80.38	164	87.34	197	520.8	41.12	76.72
DANO3MIP_LP	3202	13873	172.55	306.10	102	273.86	93	197.17	108	27.8	30.77	16.95
GEN4	1537	4297	224.72	200.34	109	144.43	98	166.33	103	41.9	645.72	66.94
LP22	2958	13434	1112.22	559.23	107	852.46	106	369.69	123	115.7	92.93	41.72
PDS-40	66844	212859	526.48	868.67	113	767.62	183	463.42	211	4754.1	358.69	345.47
RLFPRIM	58866	8052	158.43	24.26	99	20.84	116	34.30	127	28.5	12.05	3.87
STORMG2-27	14441	34114	4.07	6.28	97	4.75	120	4.42	129	72.4	8.68	1.33
STORMG2-125	66185	157496	105.85	148.97	105	103.50	143	73.99	165	2165.6	195.81	32.30
NUG08	912	1632	8.03	9.70	92	6.42	101	7.88	98	1.7	2.18	2.55
NSCT2	23003	14981	6.00	11.44	115	8.34	124	9.06	146	26.3	3.90	6.61
SGPF5Y6	246077	308634	553.72	248.91	103	168.53	129	181.51	189	7373.0	108.96	16.54
WORLD	35510	32734	965.35	644.28	143	455.90	174	316.72	190	808.7	689.91	675.82
DCP1	4950	3007	1.53	1.66	95	1.26	113	1.13	95	3.9	3.69	1.83
DCP2	32388	21087	132.04	168.66	151	76.77	196	59.27	189	257.6	375.22	118.56
DETEQ8	20678	56227	26.12	33.97	99	23.62	132	21.93	136	308.4	16.93	1.58
DETEQ27	68672	186928	301.96	349.17	120	190.95	171	142.32	183	7024.4	190.84	12.83

Larger test problem results on ***grunty*** *from § E.3.*

Bibliography

[1] (2009). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

[2] (2011). *Software Optimization Guide for AMD Family 10h and 12h Processors*. Advanced Micro Devices, Inc.

[3] (2012). *CUDA C Best Practices Guide*. NVIDIA Corporation.

[4] Adams, W. P. and Johnson, T. A. (1994). Improved linear programming-based lower bounds for the quadratic assignment problem. In *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 43–75. AMS.

[5] Al-Jeiroudi, G. (2009). *On inexact Newton directions in interior point methods for linear optimization*. PhD thesis, University of Edinburgh.

[6] Alloin, G. (1970). A simplex method for a class of nonconvex separable problems. *Management Science*, 17(1):66–77.

[7] Altman, A. and Gondzio, J. (1993). HOPDM - A higher order primal-dual method for large scale linear programming. *European Journal of Operational Research*, 66:159–161.

[8] Andersen, E. D. and Andersen, K. D. (1998). A parallel interior-point algorithm for linear programming on a shared memory machine. Technical Report Core discussion paper 9808, Center for Operations Research and Econometrics, Universite Catholique de Louvain.

[9] Aykanat, C., Pinar, A., and Çatalyürek, Ümit. V. (2004). Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 25:1860–1879.

[10] Bartels, R. H. (1971). A stabilization of the simplex method. *Numerische Mathematik*, 16:414–434.

[11] Bartels, R. H. and Golub, G. H. (1969). The simplex method linear programming using LU decomposition. *Communications of the ACM*, 12:266–268.

[12] Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation.

[13] Bennett, J. M. (1966). An approach to some structured linear programming problems. *Operations Research*, 14:636–645.

[14] Benoît, C. (1924). Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres carrés à un système d'équations linéaires en nombre inférieur à celui des inconnues. Application de la méthode à la résolution d'un système défini d'équations linéaires (procédé du Commandant Cholesky). *Bulletin Géodésique*, 2:67–77.

[15] Bieling, J. (2009). *Fast linear constraint checking through the use of GPGPU*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn.

[16] Bisseling, R. H., Doup, T. M., and Loyens, L. D. J. C. (1993). A parallel interior point algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 43:49–86.

[17] Bixby, R. E. and Martin, A. (1997). Parallelizing the dual simplex method. Technical Report CRPC-TR95706, Center for Research on Parallel Computation.

[18] Bland, R. G. (1977). New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107.

[19] Blumofe, R. D., Joerg, C. F., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. In *5^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '95*, pages 207–216.

[20] Boduroğlu, Í.. Ílkay. (1997a). *Scalable massively parallel simplex algorithms for block-structured linear programs*. PhD thesis, Columbia University.

[21] Boduroğlu, Í.. Ílkay. (1997b). Scalable massively parallel simplex algorithms for block-structured LP problems. Unpublished report.

[22] Brearley, A. L., Mitra, G., and Williams, H. P. (1975). Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83.

[23] Burkard, R. E., Çela, E., Pardalos, P. M., and Pitsoulis, L. S. (1998). The quadratic assignment problem. In *Handbook of Combinatorial Optimization*, pages 241–338. Kluwer Acadmic Publishers.

[24] Carolan, W. J., Hill, J. E., Kennington, J. L., Niemi, S., and Wichmann, S. J. (1990). An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research*, 38:240–248.

[25] Castro, J. and Cuesta, J. (2008). Quadratic regularisations in an interior-point method for primal block-angular problems. Technical Report UPC-DEIO DR 2008-07, Department of Statistics and Operational Research, Universitat Politècnica de Catalunya.

[26] Castro, J. and Frangioni, A. (2001). A parallel implementation of an interior-point algorithm for multicommodity network flows. In *Vector and Parallel Processing VECPAR 2000*, volume 1981 of *Lecture Notes in Computer Science*, pages 301–315. Springer.

[27] Çatalyürek, Ümit. V. and Aykanat, C. (1999). PaToH: A multilevel hypergraph partitioning tool, version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533, Turkey.

[28] Crowder, H. and Hattingh, J. M. (1974). Partially normalized pivot selection in linear programming. Technical Report RC 4918, IBM.

[29] Cvetanovic, Z., Freedman, E. G., and Nofsinger, C. (1991). Efficient decomposition and performance of parallel PDE, FFT, Monte Carlo simulations, simplex and sparse solvers. *The Journal of Supercomputing*, 5:219–238.

[30] Dantzig, G. and Orchard-Hays, W. (1953). Notes on linear programming: part V. - alternative algorithm for the revised simplex method using the product form of the inverse. Technical Report RM-1268, The RAND Corporation.

[31] Dantzig, G. and Wolfe, P. (1960). Decomposition principle for linear programs. *Operations Research*, 8:101–111.

[32] Dantzig, G. B. (1953). Computational algorithm of the revised simplex method. Technical Report RM-1266, The RAND Corporation.

[33] Dolan, E. and Moré, J. J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming: Series A*, 91:201–213.

[34] Drepper, U. (2007). What every programmer should know about memory. Technical report, Red Hat, Inc.

[35] Durazzi, C., Ruggiero, V., and Zanghirati, G. (2001). Parallel interior-point method for linear and quadratic programs with special structure. *Journal of Optimization Theory and Applications*, 110(2):289–313.

[36] Eckstein, J., Boduroglu, I., Polymenakos, L., and Goldfarb, D. (1995). Data-parallel implementations of dense simplex methods on the Connection Machine CM-2. *ORSA Journal on Computing*, 7:402–416.

[37] Eschermann, B. and Wunderlich, H.-J. (1990). Optimized synthesis of self-testable finite state machines. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FFTCS 20)*, pages 390–397.

[38] Ferris, M. C. and Horn, J. D. (1998). Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 80:35–61.

[39] Fitch, F. E., Munro, I., and Poblete, P. V. (1995). Permuting in place. *SIAM Journal on Computing*, 24:266–278.

[40] Forrest, J. J. (2012). Aboca: a bit of Clp accelerated? Presentation at ISMP, Berlin, 2012.

[41] Forrest, J. J. and Goldfarb, D. (1992). Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374.

[42] Forrest, J. J. H. and Tomlin, J. A. (1972). Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2:263–278.

[43] Fourer, R. (1994). Notes on the dual simplex method. Draft report.

[44] Gale, D., Kuhn, H. W., and Tucker, A. W. (1951). Linear programming and the theory of games. In Koopmans, T. C., editor, *Activity Analysis of Production and Allocation*. Wiley, New York.

[45] Gilbert, J. R. and Peierls, T. (1986). Sparse partial pivoting in time proportional to arithmetic operations. Technical report, Cornell University.

[46] Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H. (1989). A practical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474.

[47] Goldfarb, D. (2002). The simplex method for conic programming. Technical Report 2002-05, CORC, Columbia University.

[48] Goldfarb, D. and Reid, J. K. (1977). A practicable steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371.

[49] Gondzio, J. (1995). HOPDM (version 2.12) - a fast LP solver based on a primal-dual interior point method. *European Journal of Operational Research*, 85:221–225.

[50] Gondzio, J. (2012a). Interior point methods 25 years later. *European Journal of Operational Research*, 218:587–601.

[51] Gondzio, J. (2012b). Matrix-free interior point method. *Computational Optimization and Applications*, 51:457–480.

[52] Gondzio, J. and Grothey, A. (2007). Parallel interior-point solver for structured quadratic programs: application to financial planning problems. *Annals of Operations Research*, 152(1):319–339.

[53] Gondzio, J., Gruca, J., Laskowski, W., and Zukowski, M. (2012). Solving large-scale optimization problems related to Bell's Theorem. Technical Report ERGO-12-004, University of Edinburgh.

[54] Gondzio, J. and Sarkissian, R. (2003). Parallel interior-point solver for structured linear programs. *Mathematical Programming*, 96:561–584.

[55] Greef, G. (2005). The revised simplex algorithm on a GPU. Technical report, University of Stellenbosch. Cited in [15].

[56] Greenberg, H. J. and Kalan, J. E. (1975). An exact update for Harris' TREAD. *Computational Practice in Mathematical Programming*, 4:26–29.

[57] Grimes, R., Kineaid, D., and Young, D. (1979). ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas.

[58] Gruca, J., Wiesław, L., Żukowski, M., Kiesel, N., Wieczorek, W., Schmid, C., and Weinfurter, H. (2010). Nonclassicality thresholds for multiqubit states: Numerical analysis. *Physical Review A*, 82.

[59] Hall, J. A. J. (2010). Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7:139–170.

[60] Hall, J. A. J. and McKinnon, K. I. M. (1996). PARSMI, a parallel revised simplex algorithm incorporating minor iterations and Devex pricing. *Springer Lecture Notes in Computer Science*, 1184:359–368.

[61] Hall, J. A. J. and McKinnon, K. I. M. (1998). ASYNPLEX, an asynchronous parallel revised simplex algorithm. *Annals of Operations Research*, 81:27–49.

[62] Hall, J. A. J. and McKinnon, K. I. M. (2005). Hyper-sparsity in the revised simplex method and how to exploit it. *Computational Optimization and Applications*, 32:259–283.

[63] Harris, P. (1973). Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5:1–28.

[64] Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49.

[65] Ho, J. K., Lee, T. C., and Sundarraj, R. P. (1988). Decomposition of linear programs using parallel computation. *Mathematical Programming*, 42:391–405.

[66] Ho, J. K. and Sundarraj, R. P. (1994). On the efficacy of distributed simplex algorithms for linear programming. *Computational Optimization and Applications*, 3:349–363.

[67] Hogg, J. (2010). *High performance Cholesky and symmetric indefinite factorizations with applications.* PhD thesis, University of Edinburgh.

[68] Johnson, T. (2012). Generator for linearized quadratic assignment programs. `http://www.netlib.org/lp/generators/qap/`.

[69] Jones, K. L. and Lustig, I. J. (1992). Input format for multicommodity flow problems.

[70] Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on theory of computing*, pages 302–311. ACM, New York, NY.

[71] Karypis, G., Gupta, A., and Kumar, V. (1994). A parallel formulation of interior point algorithms. In *Proceedings of the 1994 ACM/IEEE conference on supercomputing*, pages 204–213. IEEE Computer Society Press, Los Amitos, CA.

[72] Karypis, G. and Kumar, V. (1994). Performance and scalability of the parallel simplex method for dense linear programming problems. Technical Report TR 94–43, University of Minnesota.

[73] Karypis, G. and Kumar, V. (1999). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20:359–392.

[74] Kaul, R. N. (1965). An extension of generalised upper bounding techniques for linear programming. Technical Report 65-27, Operations Research Center, University of California at Berkeley.

[75] Khachiyan, L. (1980). Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20:53–72.

[76] Kirilova, F. M., Gabasov, R., and Kostyukova, O. I. (1979). A method of solving general linear programming problems. *Doklady AN BSSR*, 23:197–200. Cited in [78].

[77] Knuth, D. E. (1973). *The Art of Computer Programming.* Addison-Wesley, Reading, MA.

[78] Koberstein, A. (2005). *The Dual Simplex Method, Techniques for a fast and stable implementation.* PhD thesis, Universität Paderborn.

[79] Kojima, M., Megiddo, N., and Mizuno, S. (1993). A primal-dual infeasible-interior-point algorithm for linear programming. *Mathematical Programming*, 61:263–280.

[80] Kuhn, H. W. and Quandt, R. E. (1963). An experimental study of the simplex method. In Metropolis, N., Taub, A. H., Todd, J., and Tompkins, C. B., editors, *Proceedings of Symposia in Applied Maths, Vol XV*, Providence, RI. American Mathematical Society.

[81] Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press.

[82] Lalami, M. E., Boyer, V., and El-Baz, D. (2011). Efficient implementation of the simplex method on a CPU-GPU system. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium.*

[83] Lasdon, L. S. (1970). *Optimization Theory for Large Systems.* The Macmillan Company.

[84] Lemke, C. E. (1954). The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1:36–47.

[85] Li, J., Lv, R., Hu, X., and Jiang, Z. (2011). A GPU-based parallel algorithm for large scale linear programming problem. In Watada, J., Phillips-Wren, G., Jain, L. C., and Howlett, R. J., editors, *Intelligent Decision Technologies*, volume 10 of *Smart Innovation, Systems and Technologies*, pages 37–46. Springer Berlin Heidelberg.

[86] Lubin, M., Hall, J. A. J., Petra, C. G., and Anitescu, M. (2012). Parallel distributed-memory simplex for large-scale stochastic LP problems. Submitted to Computational Optimization and Applications.

[87] Luce, R., Tebbens, J. D., Liesen, J., and Nabben, R. (2009). On the factorization of simplex basis matrices. Technical Report 09-24, Konrad-Zuse-Zentrum für Informationstechnik Berlin.

[88] Lustig, I. J. and Li, G. (1992). An implementation of a primal-dual interior point method for block-structured linear programs. *Computational Optimization and Applications*, 1:141–161.

[89] Markowitz, H. M. (1957). The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269.

[90] Maros, I. (1981). Adaptivity in linear programming, II. *Alkalmazott Matematikai Lapok*, 7:1–71.

[91] Maros, I. (1986). A general phase-I method in linear programming. *European Journal of Operational Research*, 23:64–77.

[92] Maros, I. (2003). *Computational techniques of the simplex method*. Kluwer Academic Publishers.

[93] Mehrotra, S. (1992). On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2:575–601.

[94] Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65.

[95] Mittelmann, H. (2012). Benchmarks for optimization software. `http://plato.asu.edu/bench.html`.

[96] Monteiro, R. D. C. and Adler, I. (1989). Interior path following primal-dual algorithns, part I: linear programming. *Mathematical Programming*, 44:27–41.

[97] Multicommodity problems from the Operations Research Group, University of Pisa (2012). `http://www.di.unipi.it/optimize/Data/MMCF.html/`.

[98] Nazareth, J. L. (1987). *Computer solution of linear programs*. Oxford University Press.

[99] Netlib repository at UTK and ORNL (2012). `http://www.netlib.org/`.

[100] Newton, I. (1671). *De metodis fluxionum et serierum infinitarum*.

[101] QAPLIB - A quadratic assignment problem library (2012). `http://www.seas.upenn.edu/qaplib/`.

[102] Reid, J. K. (1982). A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases. *Mathematical Programming*, 24:55–69.

[103] Roos, C., Terlacky, T., and Vial, J.-P. (2005). *Interior point methods for linear optimization*. Springer Science+Business Media, Inc.

[104] Schur, I. (1905). Neue Begründung der Theorie der Gruppencharaktere. In *Sitzungberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin*, pages 406–432. Deutsche Akademie der Wissenschaften zu Berlin.

[105] Sherman, J. and Morrison, W. J. (1949). Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Annals of Mathematical Statistics*, 20:621.

[106] Shu, W. and Wu, M.-Y. (1993). Sparse implementation of revised simplex algorithm on parallel computers. In *SIAM conference on parallel processing for scientific computing*.

[107] Sivaramakrishnan, K. K. (2008). A parallel interior point decomposition for block angular semidefinite programs. *Computation Optimization and Applications*, 46:1–29.

[108] Slater, M. (1950). Lagrange multipliers revisited: A contribution to nonlinear programming. Cowles commission discussion paper, Mathematics 403, Yale University.

[109] Smith, E., Gondzio, J., and Hall, J. A. J. (2012). GPU acceleration of the matrix-free interior point method. In *Parallel Processing and Applied Mathematics*, volume 7203 of *Lecture Notes in Computer Science*, pages 681–689.

[110] Spampinato, D. G. and Elster, A. C. (2009). Linear optimization on modern GPUs. In *International Parallel and Distributed Processing Symposium / International Parallel Processing Symposium*, pages 1–8.

[111] Suhl, L. M. and Suhl, U. H. (1993). A fast LU update for linear programming. *Annals of Operations Research*, 43:33–47.

[112] Suhl, U. H. and Suhl, L. M. (1990). Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing*, 2:325–335.

[113] Świętanowski, A. (1998). A new steepest edge approximation for the simplex method for linear programming. *Computation Optimization and Applications*, 10:271–281.

[114] Taillard, E. D. (1991). Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–455.

[115] Tebboth, J. R. (2001). *A computational study of Dantzig-Wolfe decomposition*. PhD thesis, University of Buckingham.

[116] Thomadakis, M. E. and Liu, J.-C. (1996). An efficient steepest-edge simplex algorithm of SIMD computers. In *Proceedings of the 10^{th} International Conference on Supercomputing*, pages 286–293.

[117] Tomlin, J. A. (1972). Pivoting for size and sparsity in linear programming inversion routes. *J. Inst. Maths Applics*, 10:289–295.

[118] Vázquez, F., Ortega, G., Fernández, J. J., and Garzón, E. M. (2010). Improving the performance of the sparse matrix vector product with GPUs. In *2010 10th IEEE Conference on Computer and Information Technology (CIT 2010)*, pages 1146–1151.

[119] Vuduc, R., Gyulassy, A., Demmel, J. W., and Yelick, K. A. (2003). Memory hierarchy optimizations and performance bounds for sparse $A^T Ax$. Technical Report UCB/CSD-03-1232, EECS Department, University of California, Berkeley.

[120] Wilkinson, J. H. (1961). Error analysis of direct methods of matrix inversion. *Journal of the ACM*, 8:281–330.

[121] Wolfe, P. (1959). The simplex method for quadratic programming. *Econometrica*, 27(3):382–398.

[122] Wolfe, P. (1963). A technique for resolving degeneracy in linear programming. *Journal of the Society for Industrial and Applied Mathematics*, 11:205–211.

[123] Woodbury, M. (1950). Inverting modified matrices. Technical Report Memorandum Report 42, Statistical Research Group, Princeton.

[124] Wright, S. J. (1997). *Primal-dual interior-point methods*. Society for Industrial and Applied Mathematics.

[125] Wunderling, R. (1996). *Paralleler und Objectorientierter Simplex-Algorithms*. PhD thesis, Konrad-Zuse-Zentrum für Informationtechnik Berlin.

[126] Yarmish, G. (2001). *A distributed implementation of the simplex method*. PhD thesis, Polytechnic University.

[127] Yarmish, G. and Van Slyke, R. (2009). A distributed, scalable simplex method. *The Journal of Supercomputing*, 49:373–381.

[128] Л. Г. ХАЧИЯН (1980). ПОЛИНОМИАЛЬНЫЕ АЛГОРИТМЫ В ЛИНЕЙНОМ ПРОГРАММИРОВАНИИ. *ЖУРНАЛ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И МАТЕМАТИЧЕСКОЙ ФИЗИКИ*, 20:51–68.